Introduction to EventML, version 0.2

Mark Bickford, Robert L. Constable, Richard Eaton,
David Guaspari, and Vincent Rahli

February 10, 2012

Contents

1 Introduction

1.1 Specification and Programming

EventML is a functional programming language in the ML family, closely related to Classic ML [GMW79, CHP84, KR11]. It is also a language for coding distributed protocols (such as Paxos [Ren11]) using high level *combinators* from the Logic of Events (or Event Logic) [Bic09, BC08], hence the name "EventML". The Event Logic combinators are high level specifications of distributed computations whose properties can be naturally expressed in Event Logic. The combinators can be formally translated (compiled) into the process model underlying Event Logic and thus converted to distributed programs. The interactions of these high level distributed programs manifest the behavior described by the logic. EventML can thus both specify and execute the processes that create the behaviors, called *event structures*, arising from the interactions of the processes.

Since EventML can directly specify computing tasks using the event combinators it can carry out part of the task normally assigned to a theorem prover, formal specification. EventML can also interact with a theorem prover, presently Nuprl [CAB$^+$86, Kre02, ABC$^+$06] (a theorem prover based on a constructive type theory called Computational Type Theory (CTT) [CAB$^+$86] and on Classic ML), which can express logical properties and constraints on the evolving computations as formulas of Event Logic and prove them. From these proofs, a prover can create *correct-by-construction* process terms which EventML can execute. Thus EventML and Nuprl can work together synergistically in creating a correct by construction concurrent system. EventML could play the same role with respect to any theorem prover that implements the Logic of Events. Thus EventML provides a new paradigm for creating correct distributed systems, one in which a systems programmer can design and code a system using event combinators in such a way that a theorem prover can easily express and prove logical properties of the resulting computations. To EventML, the event combinators have a dual character. They have the *logical character* of specifications and the *computational character* of producing event structures with formally guaranteed behaviors.

1.2 Interaction with theorem provers

EventML was created to work in *cooperation* with an interactive theorem prover and to be a key component of a *Logical Programming Environment* (LPE) [ABC$^+$06].

In one direction, EventML can import logical specifications from the prover as well as event class specifications and the process code that realizes them. In the present mode of operation, EventML *docks* with the Nuprl prover to obtain this information.

In the other direction, EventML can be used by programmers to specify protocols using event logic combinators. Following the line of work in which Nuprl was used to reason about the Ensemble system [Hay98, BCH$^+$00, KHH98, LKvR$^+$99] (coded in OCaml [Ler00]), EventML, by docking to Nuprl, provides a way to reason about (and synthesize) many distributed protocols. Thanks to its constructive logic, its expressiveness, and its large library, Nuprl is well suited to reason about distributed systems [BKR01]. But in principle EventML can connect to any prover that implements Event Logic and our General Process Model [BCG10]. Given an EventML specification, the Nuprl prover can: (1) synthesize process code, and (2) generate the *inductive logical form* of the specification which is used to structure logical description of the protocols and the system.

2 Event Logic

2.1 Events, event orderings, and event classes

The Logic of Events [Bic09, BC08] is a logic inspired by the work of Winskel on event structures [Win88], developed to deal with: (1) events; (2) their spatial locations; and (3) their "temporal locations," represented as a well-founded partial ordering of these events (causal order). An event is triggered by receipt of

a message; the data of the message body is called *primitive information* of the event. The Logic of Events provides ways to describe events by, among other things giving access to their associated information.

An *event ordering* is a structure consisting of: (1) a set of events, (2) a location function `loc` that associates a *location* with each event, (3) an information function `info` that associates primitive information with each event, and (4) a well-founded *causal ordering* relation on events $<$ [Lam78]. An event ordering represents a single run of a distributed system.

A basic concept in the Logic of Events is an *event class* [Bic09], which effectively partitions the events of an event ordering into those it "recognizes" and those it does not, and associates values to the events it recognizes. Different classes may recognize the same event and assign it different values. For example, one class may recognize the arrival of a message and associate it with its primitive information, the message data. Another class may recognize that, in the context of some protocol, the arrival of that message signifies successful completion of the protocol and assign to it a value meaning "consensus achieved." We specify a concurrent system in EventML by defining event classes that appropriately classify system events.

Event classes have two facets: a programming one and a logical one. On the logical side, event classes specify information flow on a network of reactive agents by observing the information computed by the agents when events occur, i.e., on receipt of messages. On the programming side, event classes can be seen as processes that aggregate information in an internal state from input messages and past observations, and compute appropriate values for them.

Formally, an event class X is a function whose inputs are event ordering and an event, and whose output is a bag of values (observations). If the observations are of type T, then the class X is called an event class of type T. The associated type constructor is $\texttt{Class}(T) = \texttt{EO} \to \texttt{E} \to \texttt{Bag}(T)$, where `EO` is the type of event orderings and `E` the type of events.

Expressions denoting events or event orderings do not occur in EventML programs; the types `EO` and `E` are not EventML types. We will refer to them when explaining the semantics of programs or reasoning about them. In particular, we will speak about the bag of values returned by a class (at some event) and will reason about the *event class relation*: we say that the class X observes v at event e (in an event ordering eo), and write $v \in X(e)$, if v is a member of $(X\ eo\ e)$. In our discussions, eo will be clear from context, so our notation omits it. If the bag of return values is nonempty we say that event e is *in* the class X, and that e is an X-event. If an event class always returns either a singleton bag or an empty bag, we call it a *single-valued* class.

Event classes are ultimately defined from one kind of primitive event class (a *base class*) using a small number of primitive *class combinators*—though users can define new combinators, and we supply a useful library of them. These primitives, and a variety of useful defined combinators are introduced in the examples of section 3. Their definitions are gathered together in section 7.

2.2 Inductive logical forms

The inductive logical form of a specification is a first order formula that characterizes completely the observations (the responses) made by the main class of the specification in terms of the event class relation. The formula is inductive because it typically characterizes the responses at event e in terms of observations made by a sub-component at a prior event $e' < e$. Such inductive logical forms are automatically generated in Nuprl from event class definitions, and simplified using various rewritings. From an inductive logical form we can prove invariants of the specification by induction on causal order.

3 Simple examples

We guide the reader through the features of this new programming/specification language with a series of examples.

Figure 1 Ping-pong protocol

```
specification ping_pong

(* ------- Imported Nuprl definitions -------  *)
import bag-map

(* ------- Protocol parameters ------- *)
parameter p : Loc
parameter locs : Loc Bag

(* ------- Interface ------- *)
input    start : Loc
internal ping  : Loc
internal pong  : Loc
output   out   : Loc

(* ------- Classes ------- *)
class ReplyToPong (client,loc) =
  let F _ l = if l = loc then {out'send client loc} else {}
  in Once(F o pong'base) ;;
class SendPing (_,loc) = Output(\l.{ping'send loc l}) ;;
class Handler (c,l) = (SendPing (c,l) || ReplyToPong (c,l)) ;;

class Delegate = (\_.\client.bag-map (\l.(client,l)) locs) o start'base;;
class P = Delegate >>= Handler;;

class ReplyToPing = (\loc.\l.{pong'send l loc}) o ping'base ;;

(* ------- Main class ------- *)
main P @ {p} || ReplyToPing @ locs
```

3.1 Ping-pong

Consider the following problem: a client wants to run some computation that involves a certain collection of nodes, but first wants to know which of them are still alive. To learn that, the client initiates the (trivial) ping-pong protocol, which will "ping" the nodes and tell the client which nodes respond to the ping. (This simple protocol does not deal with the fact that nodes can fail after responding.)

An EventML specification requires only that the declaration of an identifier precede its use. For readability, however, a specification is typically presented in the following order: name; imports (from a library); parameters; messages; variables; class declarations. Most of these parts are optional. Fig.1 presents the full EventML specification of the protocol.

Specification name

The keyword specification marks a specification's name:

```
specification ping_pong
```

Imports

EventML provides a library file that is a snapshot of Nuprl's library. The types in EventML are a subset of the types in Nuprl. Accordingly, any library function whose type is an EventML type can be used in EventML program. An import declaration makes library functions visible:

```
import bag-map
```

The bag−map operation applies a function, pointwise, to all elements of a bag:

$$\mathsf{bag{-}map}\ f\ \{a, b, \dots\} = \{(f\ a), (f\ b), \dots\}$$

Parameters

To avoid hardwiring the locations of any participants into the specification, we declare two parameters: p is the location to which clients send their requests; locs is a (non-repeating) bag containing the locations of the nodes to be checked. To execute the protocol we will instantiate those parameters as real physical machine addresses.[1] A client will identify will include its location in the request it sends, as a return address for replies.

```
parameter p : Loc
parameter locs : Loc Bag
```

Messages and directed messages

The ping-pong protocol uses four kinds of messages:

```
input    start : Loc
internal ping  : Loc
internal pong  : Loc
output   out   : Loc
```

Each of these lines declares a message *kind*. The elements of a message declaration

- identify its *category*, using one of the keywords input, output, internal

  input messages are generated by sources outside the protocol; output messages are generated by the protocol and consumed by outside sources; internal messages are produced and consumed (only) by the protocol.

- provide a (user-chosen) name for the message *kind* (in this case, start, ping, pong, or out)

- specify the type of the message body (in this case, the contents of every message exchanged in the protocol is a location).

The body of a ping or pong message will not, in fact, be an arbitrary location; it must be one of the locations in locs. However, we cannot formulate that more precise declaration of these message kinds because EventML does not allow subtype definitions (though Nuprl does).

Our discussions will use the notation [start : x] to denote a start message with body x, etc. [2]

A *directed message* is a pair consisting of a location (the addressee) and a message. Our discussions will use the notation (loc, msg) to denote the directed message that addresses message msg to location loc. Directed messages have a special semantics. When a *main* class (see below) produces a bag of directed messages, a messaging system attempts to deliver them—i.e., given the directed message (loc, msg), the messaging system attempts to deliver msg to location loc. We reason about the effect of a protocol under assumptions about message delivery. For present purposes, we assume that all messages are eventually delivered at least once, but make no assumption about transit times or the order in which messages are delivered.

Message declarations automatically introduce certain operations and event classes:

[1] As a logical matter, an EventML program may have parameters of any type definable in EventML. To compile an EventML specification, a developer must supply a configuration file that instantiates the parameters. See section 8.

[2] For technical reasons, the Nuprl model represents a message as a triple: a list of tokens acting as a message header, the message body, and the type.

- Every declaration of an input or internal message kind foo automatically introduces a *base* class, denoted foo'base, an event class that recognizes the arrival of a foo message and observes its body (and recognizes messages of no other kind).

  More precisely, the arrival of a message [foo : *msg*] at location l, causes an event e to happen at l; and when e occurs, foo'base observes the content of that message. (Equivalently, we say that foo'base returns $\{msg\}$; or we say that $v \in \mathsf{foo'base}(e)$ if and only if $v = msg$.)

- Every declaration of an output or internal message kind foo automatically introduces functions foo'send and foo'broadcast that are used to construct directed messages or bags of them. (These are the only way to construct such messages—thus, we assume that the messages they construct cannot be forged.)

  If l is a location and *msg* is of the appropriate type

  $$(\mathsf{foo'send}\ l\ msg) = (l, [\mathsf{foo} : m])$$

  the directed message for sending [foo : m] to l.

  If the l_i are locations and *msg* is of the appropriate type

  $$\mathsf{foo'broadcast}\ \{l_1, \ldots, l_n\}\ msg = \{(l_1, [\mathsf{foo} : msg]), \ldots, (l_n, [\mathsf{foo} : msg])\}$$

  a bag of directed messages containing a [foo : *msg*] message for each location l_i.

Ultimately, all EventML programs are defined by applying combinators to base classes, which are the only primitive classes.[3] We assume that any computing system on which we wish to implement EventML provides the means to implement base classes.

The protocol

The ping-pong protocol proceeds as follows:

1. It begins when a message of the form [start : *client*] arrives at location p. (Replies to this message will be sent to *client*.)
2. A supervisory class, P, will then spawn several classes at location p. For each l in locs, it spawns the class Handler(*client*,l), which will handle communications with node l.
3. Handler(*client*, l) sends a [ping:p] message to the node at location l and waits for a response.
4. On receipt of a [ping:p] message, the ReplyToPing class at node l sends a [pong : l] message back to p.
5. On receipt of this [pong : l] message, Handler(*client*,l) sends an [out : l] message to *client* and terminates.

Intuitively, Handler is a parameterized class—but, because EventML is a higher-order language, we need no special generic or template construct in order to express that. An event class parameterized by values of type T is simply a function that inputs values of type T and outputs event classes.

Class combinators

Our specification uses the following class combinators, all of them provided in the standard library:

- Output(f): If f : Loc $\to$ Bag(T), Output(f) is the class that, in response to the first event it sees at location l, returns the bag of values (f l); it then terminates.
- X || Y: This event class is the parallel composition of classes X and Y. It recognizes events in either X or Y, and observes a value iff either X or Y observes it. The parallel combinator is a primitive.

[3] Technically, this is not quite true: It would be possible to define an event class in the Nuprl library and import it into an EventML program.

- X >>= Y: This is the *delegation*, or *bind*, combinator.[4] If X is an event class and Y is a function that returns event classes, X >>= Y is the event class that, whenever it recognizes an event, acts as follows: For each $v \in \mathsf{X}(e)$, it "spawns" the class (Y v). (Events in this spawned class will occur at the same location as e and causally after it. We temporarily defer a more precise discussion of its meaning.) Delegation is primitive.

- f o X: This is well typed if f is a function that takes as arguments a location and a value, and returns a bag. It acts as follows (for the case of a single-valued class): when $v \in \mathsf{X}(e)$and $NUPRLevent$ has location loc, f o X returns (f loc v). This *simple composition* combinator is "almost primitive." It is defined in terms of a primitive combinator that is somewhat more expressive but rarely, if ever, used. (This combinator is described in more detail in the discussion, below, of class ReplyToPong.)

- Once(X): This class responds only to the first X-event at any location and, at such an event, behaves like X. That is, $v \in (\mathsf{Once}(\mathsf{X}))(e)$ iff $v \in \mathsf{X}(e)$ and there was no X-event prior at location $\mathtt{loc}(e)$ prior to e. Once is a defined combinator that our compiler treats specially, because it knows that the process Once(X) can be killed and cleaned up after it has recognized one event.

- X@b: This is the restriction of X to the locations in the bag b: $v \in (\mathsf{X@b})(e)$ iff e occurs at a location in b and $v \in \mathsf{X}(e)$. Operationally, it means "run the program for X at each location in b."

The "main" class

The keyword main identifies the event class that compilation of an EventML specification will actually implement (given appropriate instantiations of its parameters). The main program of the ping-pong program is the parallel composition of the supervisory class P running at location p and ReplyToPing running at all the locations in locs:

```
main P @ {p} || ReplyToPing @ locs
```

Spawning of handlers (delegation to sub-processes)

The supervisory class P uses the delegation combinator to spawn a handler for each request.

```
class P = Delegate >>= Handler;;
```

We will define Delegate so that, in response to [start : x], returns the bag $\{(\mathsf{x}, l_1), (\mathsf{x}, l_2), \ldots\}$, where the l_i are the elements of locs. Each element of this bag is the initial data needed by one of the spawned classes. The effect of Delegate >>= Handler is therefore to spawn a class (Handler $(client, l_i)$) for each i. Notice how the types match up: Delegate is an event class of type Loc * Loc; Handler will be a function mapping values of type Loc * Loc to event classes of type directed message. Therefore Delegate >>= Handler is an event class that returns (bags of) directed messages.

Consider the definition of Delegate, which we've rewritten by introducing the locally defined function f.

```
class Delegate =
   let f = \client.bag-map (\l.(client,l)) locs in
      (\_.f) o start'base ;;
```

The "_" is used, as in ML, as the name of a variable whose value is ignored.

When a [start : x] message arrives, we want Delegate to return the bag

$$\mathsf{f}\ x = \{\langle x, l_1\rangle, \langle x, l_2\rangle, \ldots\}$$

where locs $= \{l_1, l_2, \ldots\}$. Intuitively, the simple composition operator transforms observed values by applying a function. However, the function we use is not f but _.f. The reason is that the location

[4] The class type forms a monad and delegation is the bind operator of that monad.

of an event is also observable; accordingly, we define "g o X" so that that g takes *two* arguments: the location of the event and a value observed by X. It so happens that in this case, the location is ignored. We give a precise definition of this combinator at the end of this section.

Handler

Interactions betwen the Handler classes spawned by P and the nodes carry out steps (3)–(5) of the protocol. The input to the higher-type function Handler is a pair of locations: the client location and the location to ping. The resulting handler is the parallel composition of two other parameterized classes: SendPing, which executes step (3) of the protocol, and ReplyToPong, which executes step (5).

```
class Handler (c,l) = (SendPing (c,l) || ReplyToPong (c,l)) ;;
```

By the definition of the parallel combinator, Handler (c, l) computes everything that either SendPing (c, l) or ReplyToPong (c, l) does.

SendPing (c, l) is in charge of only one task: send a ping message to l.

```
class SendPing (_,loc) = Output(\l.{ping'send loc l}) ;;
```

By the definitions of Output and ping'send given above, an instance of SendPing$(client, loc)$ running at location l will respond to the first event it sees at l by directing a ping message with body l to location loc; it will then terminate. The recipient will interpret l as a return address.

ReplyToPong $(client, loc)$ waits for a pong message from the node at location loc and, on receiving one, sends [out : l] to location $client$. It therefore responds to a *subset* of the events recognized by the base class pong'base: not every pong message, but only those sent from loc, i.e., those whose message body is loc. If $v \in$ pong'base(e), ReplyToPong generates an output by applying the following function to v:

```
\l.if l = loc then {out'send client loc} else {}
```

and then terminates. This is the essence of the locally defined function F in

```
class ReplyToPong (client,loc) =
  let F _ l = if l = loc then {out'send client loc} else {}
  in Once(F o pong'base) ;;
```

Once again, because the response doesn't depend on the location at which the input event occurred, the first argument to F is a dummy.

ReplyToPing

ReplyToPing defines a program that must run at each node that will be pinged.

```
class ReplyToPing = (\loc.\l.{pong'send l loc}) o ping'base ;;
```

This time the transformation function makes use of the initial location argument. When the class ((\loc.\l.{pong'send l loc}) o ping'base), running at location s, receives [ping : l] it sends [pong : s] to location l.

Programmable classes

No base class can be a main program. For example, start'base recognizes the arrival of every start message at any node whatsoever; but this abstraction cannot be implemented: we cannot install the necessary code on every node that exists (whatever that may mean). However, start'base is *locally programmable* in the sense that we can implement any class that results from restricting it to a finite set of locations. All base classes are locally programmable and all primitive class combinators preserve

the property of being locally programmable, so every class definable in an EventML program is locally programmable.[5]

A class is *programmable* if it is equivalent to the restriction of a locally programmable class to a finite set of locations. Declaring a class as a main program incurs the obligation to prove that it is programmable. Using the "_@_" combinator, and the fact that primitive combinators also preserve the property of being programmable, we can automatically prove that any idiomatically defined main program is programmable.

Simple composition, in detail

One can apply simple composition to any number of classes. Given n classes $X_1, \ldots, X_n$, of types $T_1, \ldots, T_n$ respectively, and given a function F of type $\texttt{Loc} \to T_1 \to \cdots \to T_n \to \texttt{Bag}(T)$, one can define a class $C : \texttt{Class}(T)$ by $C = F \texttt{ o } (X_1, \cdots, X_n)$.

Intuitively, C processes an event e as follows. The first argument supplied to F is the location at which e occurs; the successive arguments are, in order, the values observed by the classes X_i at e; and C returns the bag that F computes from these inputs. That description leaves it unclear what to do if, for some i, e is not an X_i-event, or what to do if for some i, X_i produces a bag with more than one element.

Here is a more precise formulation. C produces (observes) the element v of type T iff each class X_i observes an element v_i of type T_i at event e and $v = f\ \texttt{loc}(e)\ v_1\ \cdots\ v_n$. Therefore, a C-event must be an X_i-event for all $i \in \{1, \ldots, n\}$. If for some $i \in \{1, \ldots, n\}$, X_i does not observe anything at event e, then neither does C.

3.2 Ping-pong with memory

We now make our ping-pong protocol a bit more interesting by adding some memory to the main process. We introduce a new integer parameter, threshold; instead of sending an [out : l] message to the client whenever node l responds to a pong, we wait until a total of threshold responses have been received, and then notify the client by sending a message [out : $[l_1; l_2; \ldots; l_{threshold}]$], whose body is the list of all responders. We modify the design of ping-pong by adding one more (parameterized) class, a memory module: Instead of sending an out message directly to a client, ReplyToPong will send an alive message to an appropriate memory module, which will accumulate responses and send an out message to the client once it has received enough of them.

And we add one more twist. A client who sends multiple start messages will receive multiple out messages in reply and may need to know what request any out message is replying to. So the client will attach an integer id (we will call it a *request number*) to its start messages; that request number will be included in the out message it receives. The request numbers need not be globally unique identifiers, so we will also arrange for the supervisory class P to attach a global id (which we will call a *round*) to each request that it receives. The protocol proceeds as follows:

1. P receives a [start : $\langle client, req\_num \rangle$] message from the location *client*.
2. P generates a unique id, *round*, for the request and spawns the following:
   - for each node l in locs, a class Handler(l, *round*)
   - a memory module (Mem *client req_num round*)
3. Handler(l,*round*) sends a [ping:$\langle$p, *round*$\rangle$] message to the node at location l and waits for a reply.
4. On receipt of [ping:$\langle$p, *round*$\rangle$] the ReplyToPing class at node l sends a [pong:$\langle$l, *round*$\rangle$] message to p. Handler classes respond to pong messages.
5. On receipt of [pong:$\langle l, round \rangle$], the class Handler(l, *round*) sends an [alive:$\langle l, round \rangle$] message to itself (location p). Mem classes respond to alive messages.

[5] This needs a qualification: One could define a class in Nuprl that is not locally programmable and then import it from the Nuprl library; one could similarly introduce a pathological combinator. If that is done, the first step of compilation—verifying that the main program is programmable—would fail.

Figure 2 Ping-pong protocol with memory

```
specification m_ping_pong

(* ——— Imported Nuprl declarations ——— *)
import bag-map deq-member length

(* ——— Parameters ——— *)
parameter p         : Loc
parameter locs      : Loc Bag
parameter threshold : Int

(* ——— Interface ——— *)
input    start : Loc * Int
internal ping  : Loc * Int
internal pong  : Loc * Int
internal alive : Loc * Int
output   out   : Loc List * Int

(* ——— Classes ——— *)
class ReplyToPong p =
  let F slf q = if p = q then {alive'send slf p} else {}
  in F o pong'base ;;
class SendPing (loc,round) = Output(\l.{ping'send loc (l,round)}) ;;
class Handler p = SendPing p || ReplyToPong p ;;

class MemState round =
  let F _ (loc:Loc,r:Int) L =
    if r = round & !(deq-member (op =) loc L) then {loc.L} else {L}
  in F o (alive'base,Prior(self)?{[]});;
class Mem client req_num round =
  let F _ L = if length L >= threshold then {out'send client (L,req_num)} else {}
  in F o (MemState round);;

class Round (client, req_num, round) =
    (Output(\_.locs) >>= \l.Handler (l,round))
  || Once(Mem client req_num round);;

class PState =
  let F loc (client,req_num) (_,_,n) = {(client, req_num, n + 1)}
  in F o (start'base,Prior(self)?(\l.{(l,0,0)}));;
class P = PState >>= Round;;

class ReplyToPing = (\loc.\(l,round).{pong'send l (loc,round)}) o ping'base ;;

(* ——— Main class ——— *)
main P @ {p} || ReplyToPing @ locs
```

6. When (Mem *client req_num round*) has seen alive messages from *threshold* distinct locations, it sends to location *client* an appropriate out message tagged with *req_num*.

Fig. 2 provides the full specification of this protocol. Most of it is a routine adaptation of the ping-pong specification. The novelty lies in the introduction of the event classes PState and Mem that act like state machines. We will describe these in detail.

Imported library functions

The specification imports two Nuprl functions.

- length, which computes the length of a list
- deq−member, which checks whether an element belongs to a list

  To apply this to lists of type T we must also supply an operation that decides equality for elements of T. That operation is a parameter to the membership test; thus, we write (deq−member eq y lst) to compute the value of the boolean "y is a member of list lst, based on the equality test eq."

Class combinators

The specification uses the three remaining primitive combinators:

- Prior(X): Event e belongs to Prior(X) if some X-event has occurred at $\texttt{loc}(e)$ strictly before event e; if so, its value is the value returned by X for the most recent such X-event. Once an X-event has occurred at location l, all subsequent events at l are Prior(X)-events.
- X?f: For any class X of type T, and any function $\texttt{f} : \texttt{Loc} \to \texttt{Bag}(\texttt{T})$, X?f has the following meaning:

$$v \in (\texttt{X?f})(e) \qquad \text{iff} \qquad \begin{cases} v \in \texttt{X}(e) & \text{if } e \text{ is an X-event} \\ v \in \texttt{f}(\texttt{loc}(e)) & \text{otherwise} \end{cases}$$

  If (f l) is nonempty, then all events at location l are (X?f)-events.

- self: The underlying semantic model of EventML has powerful operators for defining event classes by recursion, including mutual recursion. However, EventML itself currently provides only a simple recursion scheme, which has been adequate for all the practical examples we have considered. The keyword self can occur only in contexts such as

  ```
  class X = G (Prior(self)?f)
  ```

  where, instead of being simply an argument to a function, Prior(self)?f could be a subterm of a more general expression. As a result of this definition X satisfies the fix-point equation

  ```
  X = G (Prior(X)?f)
  ```

  that specifies the value of X at any event e in terms of its value at the immediately prior X-events; or, if there is no prior X-event, in terms of $\texttt{f}(\texttt{loc}(e))$. Examples will make this clear.

P and PState

Class P uses PState to generate a unique round number for each request, and passes that to Round, which in turn performs step 2 of the protocol. The definition of PState is recursive.

```
class PState =
  let F loc (client,req_num) (_,_,n) = {(client, req_num, n + 1)}
  in F o (start'base, Prior(self)?(\l.{(l,0,0)}));;
```

This defines a state machine as follows:

- start'base-events trigger change of state.
- The state type of PState is (Loc ∗ int ∗ int). The state components represent, respectively: the client whose request has caused the state change, the request number assigned by the client, and the most recent round number generated by PState.
- For any start'base-event e, $v \in \texttt{PState}(e)$ iff v is the state of PState after it has processed event e.

- The initial value of the state at location l is $(l, 0, 0)$.

  The first two components of this initial state are dummy values.

- The transition function at location l is (F l).

  If [start:$\langle client, req\_num\rangle$] arrives in state (l, r, n), the new state is $(client, req\_num, n+1)$.

As an exercise, we unroll some instances of the definition. By definition, PState satisfies the recursion equation:

```
PState =
 let F loc (client,req_num) (_,_,n) = {(client, req_num, n + 1)}
 in F o (start'base,Prior(PState)?(\l.{(l,0,0)}));;
```

Note first that, because the return value of

```
\l.{(l,0,0)}
```

is always nonempty, every event belongs to the class

```
Prior(PState)?(\l.{(l,0,0)})
```

It follows from this that the PState-events are precisely the start'base-events. (The locally defined function F always returns a nonempty result; therefore, for any A and B, the events in F o (A,B) will be those events that are both A-events and B-events.)

Suppose that event e_1, the arrival of the message [start:$\langle c_1, r_1\rangle$], is the first PStart-event occurring at location l. Call it event e_1. At e_1, PState returns

$$\text{F } l\ (c_1, r_1)\ (l, 0, 0) = \{(c_1, r_1, 1)\}$$

Suppose e_2, the arrival of the message [start:$\langle c_2, r_2\rangle$], is the next PStart-event occurring at location l. At e_2, PState returns

$$\text{F } l\ (c_2, r_2)\ (c_l, r_1, 1) = \{(c_2, r_2, 2)\}$$

The key point is that the argument supplied to F by

```
Prior(PState)?(\l.{(l,0,0)})
```

is the value of the state when the incoming message *arrives*—which is the value returned as a result of the previous start message—or, if there hasn't been one, $(l, 0, 0)$.

Mem and MemState

The state machine PState maintains an internal state and after an input event returns a singleton bag containing its new state. It is, essentially, a Moore machine.

A memory module will maintain an internal state (listing the nodes from which alive messages have been received); it outputs not its state but an out message—and not every change of state will cause an output. A simple way to achieve this is to define two classes: MemState, like PState, simply accumulates a state and makes it visible; Mem observes MemState and generates an output when appropriate.

The class (MemState *round*) accumulates and makes visible the internal state:

```
class MemState round =
  let F _ (loc:Loc,r:Int) L =
    if r = round & !(deq-member (op =) loc L)
    then {loc.L}
    else {L}
  in F o (alive'base,Prior(self)?{\l.[]});;
```

An input event to this state machine is the arrival of an alive message. The state is a list of locations, initially empty; it contains the distinct locations from which alive messages have been received for round number *round*, and ignores all other messages. When a message arrives with body (loc, r) the new state is determined as follows: if the message's round number is *round*, and *loc* is not yet on the list, prepend *loc* to the state; otherwise, no change. (Because round numbers are globally unique, this class can perform its function without knowing either the client who initiated the request or the request number assigned.)

Notation: Some of the formal arguments to the function F are labeled with types: (loc:Loc,r:Int), rather than (loc,r). It is always legal to label patterns or expressions with types; and, in some situations, the type inference algorithm needs the extra help. The use of labels can be eliminated by using variable declarations, which are introduced in section 5.

Notation: Recall that the first argument to deq−member must be an equality operation. In the term "deq−member (op =) loc L" the equality operation is denoted by "(op =)." In general, "(op g)" means "g used as a binary infix operator."

When (Mem client req_num round) sees that the state of (MemState round) has grown to a list of length threshold it signals the client.

```
class Mem client req_num round =
  let F _ L = if length L >= threshold
              then {out'send client (L,req_num)}
              else {}
  in F o (MemState round);;
```

3.3 Leader election in a ring

Many distributed protocols require that a group of nodes choose one of them, on the fly, as a leader. Here is a simple strategy for doing that under the assumptions that:

- the nodes are arranged in a ring (each node knowing its immediate successor)
- each node has a unique integer id

Any node may start an election by sending its own id to its immediate successor (a *proposal*). With one exception, a node that receives a proposal will forward to its successor the greater of the following two values: {the proposal it received, its own id}. The exception occurs if (and only if) a node receives in a proposal its own id. In that case, the node stops forwarding messages and declares itself elected. If messages are delivered reliably and no nodes fail, this protocol will always succeed in electing the node with the greatest id.

Fig. 3 presents our specification of a slightly more sophisticated protocol. We add an interface that makes it possible for some external party to reconfigure the ring—e.g., if it believes that some nodes have failed. Informally, we call the intervals between reconfigurations *epochs* (setting aside the vagueness of "between" in a distributed setting). We number the epochs with positive integers—using 0 to mean "no epoch has started at this node."

The inputs to the protocol are of two kinds:

- a config message tells a node to begin a new epoch and stipulates which node is, in the new epoch, its immediate successor in the ring;
- a choose message contains the number of an epoch, and asks for an election in that epoch.

The outputs of the protocol are leader messages sent to some designated client. The body of a leader message contains an epoch number and the id of the leader elected in that epoch.

Parameters to the protocol are

- nodes : Loc Bag – the nodes from which a leader must be chosen

Figure 3 Leader election in a ring

```
specification leader_ring

(* ------- Parameters ------- *)
parameter nodes  : Loc Bag
parameter client : Loc
parameter uid    : Loc → Int

(* ------- Imported Nuprl declarations ------- *)
import imax

(* ------- Type functions ------- *)
type Epoch = Int

(* ------- Interface ------- *)
input    config  : Epoch * Loc (* To inform a node of its Epoch and neighbor *)
output   leader  : Epoch * Loc (* Location of the leader    *)
input    choose  : Epoch       (* Start the leader election *)
internal propose : Epoch * Int (* Propose a node as the leader of the ring *)

(* ------- Classes ------- *)
let dumEpoch = 0 ;;

class Nbr =
  let F _ (epoch, succ) (epoch', succ') =
    if epoch > epoch'
    then {(epoch, succ)}
    else {(epoch', succ')}
  in F o (config'base,Prior(self)?(\l.{(dumEpoch,l)})) ;;
class PrNbr = Prior(Nbr)?(\l.{(dumEpoch,l)}) ;;

class ProposeReply =
  let F loc (epoch, succ) (epoch', ldr) =
    if epoch = epoch'
    then if ldr = uid loc
         then {leader'send client (epoch, loc)}
         else {propose'send succ (epoch, imax ldr (uid loc))}
    else {}
  in F o (PrNbr,propose'base) ;;

class ChooseReply =
  let F loc (epoch, succ) epoch' =
    if epoch = epoch'
    then {propose'send succ (epoch, uid loc)}
    else {}
  in F o (PrNbr,choose'base) ;;

(* ------- Main class ------- *)
main (ProposeReply || ChooseReply)  @ nodes
```

- client : Loc – the node to be informed of the election results
- uid : Loc → Int – a function assigning a unique id to each member of nodes

Our slightly generalized protocol is still quite simple to describe. A node keeps track of the epoch in which it is currently participating and ignores all propose or choose messages labeled with other epochs. If it receives a config message for an epoch numbered higher than its current epoch, it switches to the new epoch, and otherwise ignores it. A node reacts to all non-ignored propose and choose messages as in the original protocol.

The delicate part lies in formulating the invariants preserved by the protocol and the conditions under which it succeeds. What if reconfiguration occurs while an election is going on? What if config messages arrive out of order—requesting epoch 4 and later requesting epoch 3? What if config messages partition the nodes into two disjoint rings? We ignore those questions.

Nbr, the state of a node

Informally, the state of any node is a pair $\langle epoch, succ \rangle$: Int $*$ Loc, where *epoch* is the number of its current epoch and *succ* is the location of its current successor. This state changes only in response to config messages. We capture that behavior in the class Nbr, which defines a state machine as follows:

- At location l, its initial state is $\langle 0, l \rangle$; essentially, these are both dummy values.
- Input events are the arrivals of config messages, which are recognized by the base class config'base.
- The state transition in response to the input $\langle epoch', succ' \rangle$ is: if $epoch' > epoch$, then change to $\langle epoch', succ' \rangle$; otherwise, no change.

We use the state machine idiom described in section 3.2. In addition to Nbr, which observes the state after an input has been processed, we define PrNbr, which observes the state when an input arrives and before it has been processed. (In fact, Nbr is only an auxiliary for the sake of defining PrNbr.)

```
let dumEpoch = 0 ;;

class Nbr =
  let F _ (epoch, succ) (epoch', succ') =
    if epoch > epoch'
    then {(epoch, succ)}
    else {(epoch', succ')} in
  F o (config'base, Prior(self)?(\l.{(dumEpoch, l)})) ;;
class PrNbr = Prior(Nbr)?(\l.{(dumEpoch, l)}) ;;
```

Factoring the main program.

We factor the behavior of the protocol into two classes, one triggered by propose messages and one triggered by choose messages. We define

```
main (ProposeReply || ChooseReply)  @ nodes
```

and will define both ProposeReply and ChooseReply in terms of PrNbr.

ProposeReply.

The response to a proposal is as described informally: send a leader message if you receive your own id; otherwise, propose to your successor the max of the proposal received and your own id.

```
class ProposeReply =
  let F loc (epoch, succ) (epoch', ldr) =
    if epoch = epoch'
    then if ldr = uid loc
         then {leader'send client (epoch, loc)}
         else {propose'send succ (epoch, imax ldr (uid loc))}
    else {}
  in F o (PrNbr, propose'base) ;;
```

Since Nbr changes only in response to config messages, the state of Nbr is the same both before and after a propose message arrives. So why couldn't we simplify this definition by replacing the expression "F o (PrNbr,Propose)" with "F o (Nbr,Propose)"?

The reason is that Nbr can only observe config'base-events, whereas PrNbr can observe any event e. This use of (Prior (...))?(...) is a basic idiom of EventML programming—although, as will be seen in section 4, it is often conveniently packaged within standard library combinators.

Note: If e is a propose'base-event at location loc, and no config'base-event has yet occurred at loc, then e is a PrNbr-event, and the only value PrNbr observes at e is the pair (dumEpoch,loc).

ChooseReply

When ChooseReply receives a choose instruction for the epoch on which it is currently working, it initiates an election by sending an appropriate propose message.

```
class ChooseReply =
  let F loc (epoch, succ) epoch' =
    if epoch = epoch'
    then {propose'send succ (epoch, uid loc)}
    else {}
  in F o (PrNbr,choose'base) ;;
```

This uses PrNbr instead of Nbr for the same reason that ProposeReply does.

4 State machines

Previous examples have built state machine classes by hand, from EventML primitives. The Nuprl library defines combinators that package up idioms for defining state machines of various kinds. Many of the automated tactics created to reason about event classes are tuned for definitions that use these combinators.

4.1 Moore machines, "pre" and "post"

We have been using a standard strategy. First define what might loosely be called a Moore machine: in response to inputs it updates its state and makes that state visible. We then use the simple composition combinator to define a Mealy machine (loosely called) from this Moore machine: one that, in response to some of the Moore machine's inputs, returns directed messages. One virtue of this factoring is that, by making the state visible, we can conveniently express state invariants as properties of classes explicitly defined in the EventML code.

In general, we can define a Moore machine from the following data:

- A, the type of input values
- S, the type of state values
- X : Class(A), recognizing input events
- $init$: Loc $\to$ Bag(S), assigning a bag of initial states to each location
- tr : Loc $\to A \to S \to S$, assigning a transition function to each location

We introduce combinators, SM1−class and Memory1, that provide two different ways to observe this state machine. In the idiomatic case, in which $init$ assigns a singleton bag to every location:

- (SM1−class $init$ (tr, X)) is the "post" observer of the state machine, which behaves as follows:
  - The events it recognizes are the X-events.

  - To every X-event e it assigns a singleton bag $\{v\}$, where v is the state of that state machine after responding to e.

- (Memory1 *init tr X*) is the "pre" observer of the state machine, which behaves as follows:[6]
  - It recognizes *all* events.
  - To every event e it assigns a singleton bag, $\{v\}$ where, intuitively, v is the value of the state when e arrives (*before* it is processed).
    More precisely, if there has been no previous X-event at location $\texttt{loc}(e)$, $\{v\} = (\mathit{init}\ \texttt{loc}(e))$; otherwise, letting e' be the most recent such X-event before e, $\{v\} = (\mathsf{SM1{-}class}\ \mathit{init}\ (tr, X))(e')$

We can define[7] these combinators as follows:

```
class SM1–class init (tr,X) = tr o (X, Prior(self)?init) ;;
class Memory1 init tr X = Prior(SM1–class init (tr,X))?init ;;
```

Thus, if we declare

```
class Y = SM1–class init (tr,X);;
class PrY = Memory1 init tr X;;
```

we know that that the following equations are satisfied:

```
Y = tr o (X, Prior(Y)?init)
PrY = Prior(Y)?init
```

4.2 Moore machines with multiple transition functions

One often wants a state machine whose inputs are defined by two or more different classes—typically, base classes that recognize inputs of different kinds. For notational simplicity, consider the case of two input classes. Now we have, for $i = 1, 2$:

- A_i, a type of input values
- S, a type representing values of the state
- $X_i : \texttt{Class}(A_i)$ recognizing input events
- $\mathit{init} : \texttt{Loc} \to \texttt{Bag}(S)$, assigning a bag of initial states to each location
- $tr_i : \texttt{Loc} \to A_i \to S \to S$, assigning transition functions to each location

Together with *init*, each of the pairs $\langle tr_i, X_i \rangle$ defines a state machine with the same state type, S, but possibly different types of input values. In the idiomatic case, the X_1-events and the X_2-events are disjoint and the state machine we want to define acts as follows: If e is an X_i-event, it takes the transition defined by tr_i. (The definition will guarantee that if e should be both an X_1-event and an X_2-event, the state machine takes transition tr_1.)

As before, we can define classes that represent both "post" and "pre" observations of this state machine (for the idiomatic case):

- (SM2–class *init* (tr_1, X_1) (tr_2, X_2)) is the "post" observer. The events it recognizes are X_1-events or X_2-events.

[6] The use of two separate parameters, *tr* and X, rather than a single pair, is a slightly awkward bit of legacy that will eventually be changed.

[7] The definition we use for SM1–class is not literally this one, but is equivalent to it.

Figure 4 2/3 consensus: preliminaries

```
specification rsc

(* ------- Parameters ------- *)
(* consensus on commands of arbitrary type Cmd with equality decider (cmdeq) *)
parameter Cmd, cmdeq : Type * Cmd Deq
parameter coeff      : Int
parameter flrs       : Int      (* max number of failures                  *)
parameter locs       : Loc Bag (* locations of (3 * flrs + 1)   replicas  *)
parameter clients    : Loc Bag (* locations of the clients to be notified *)

(* ------- Imported Nuprl declarations ------- *)
import length poss-maj list-diff deq-member from-upto bag-append Memory1

(* ------- Type definitions ------- *)
type Inning   = Int
type CmdNum   = Int
type RoundNum = CmdNum * Inning
type Proposal = CmdNum * Cmd
type Vote     = (RoundNum * Cmd) * Loc

(* ------- Messages ------- *)
internal vote    : Vote
internal retry   : RoundNum * Cmd
internal decided : CmdNum
output   notify  : Proposal
input    propose : Proposal

(* ------- Variables ------- *)
variable sender  : Loc
variable loc     : Loc
variable ni      : RoundNum
variable n       : CmdNum

(* ------- Auxiliaries ------- *)
let init x loc = {x} ;;
```

- (Memory2 *init* tr_1 X_1 tr_2 X_2) is the "pre" observer, which recognizes *all* events.

SM3−class/Memory3 and SM4−class/Memory4 are similar, except that they combine, respectively, three and four different sources of inputs.

Note: EventML is rich enough to define all of these classes. For technical reasons, their official definitions use features of the Nuprl type system not available in EventML, so we prefer simply to make instances of these combinators available as quasi-primitives.

5 The two-thirds consensus protocol

Consider the following problem: A system has been replicated for fault tolerance. It responds to commands issued to any of the replicas, which must come to consensus on the order in which those commands are to be performed, so that all replicas process commands in the same order. Replicas may fail. We assume that all failures are crash failures: that is, a failed replica ceases all communication with its surroundings. The two-thirds consensus protocol is a simple protocol for coming to consensus, in a manner

that tolerates n failures, by using (precisely) $3n + 1$ replicas.

Input events communicate *proposals*, which consist of integer/command pairs: $\langle n, c \rangle$ proposes that command c be the n^{th} one performed. The protocol is intended to obtain agreement, for each n, on which command will be the n^{th} to be performed, and to broadcast a notify message with those decisions (which are also integer/command pairs) to a list of clients.

Each copy of the replicated system will contain a module that carries out the consensus negotiations. In this specification we describe only those modules (which we continue to call Replicas). To specify the full system we would have to include a description of how those decisions are used. That is done in the description of the Paxos protocol (section 6).

For convenient display, we split the full specification into smaller chunks: figure 4 contains the prefatory information (parameters, imports, type definitions, message declarations, variables, auxiliaries) and figures 5 through 8 define the classes. Section 5.1 walks through code, redisplaying fragments of the text as they are discussed. A reader may find it helpful first to concentrate on the informal description of each class provided and then, before studying details, turn to section 5.2 to see some scenarios showing the protocol in action. Section 5.3 explains why the protocol satisfies the basic safety property of *consistency*—it will not send contradictory notifications. That section also defines the precise sense in which the protocol can "tolerate" up to flrs "failures," but does not provide a proof of that.

5.1 The specification of 2/3-consensus

5.1.1 Preliminaries

This section comments on the preliminary definitions given figure 4, and also introduces the library combinator until.

Parameters

The parameters of the protocol are

- Cmd: the type of commands
- flrs : the max number of failures to be tolerated
- locs: the locations of the $3 *$ flrs $+1$ replicated processes that decide on consensus
- clients : the locations of the clients to be notified of decisions

We make no assumptions about who submits inputs or constraints on how they are submitted.

The declaration of the Cmd parameter also introduces a parameter for an equality operator:

```
parameter Cmd, cmdeq : Type * Cmd Deq
```

When we instantiate the type Cmd, we must also instantiate cmdeq with an operation that decides equality for members of that type. The keyword Deq denotes a type constructor: (Cmd Deq) is the type of all equality deciders for Cmd. We need cmdeq because we want to apply deq−member to compute membership in a list of commands; as noted in section 3.2, we must therefore supply an equality decider.

Variables

One reason for the variable declarations, such as

```
variables sender : Loc
variable ni      : RoundNum
```

is to introduce notational conventions that make the specification easier to read. Type checking will object if the notations are misused.

A second reason is to help the type inference algorithm, which sometimes requires hints about the types of the arguments to functions being defined. An expression or pattern may be labeled with a type,

which will be checked statically, and may also constrain polymorphism that might otherwise arise. E.g., after

```
let foo x = x ;;
let bar (x:Int) = x ;;
let baz (x,y:Bool) = (x,y) ;;
```

foo is thepolymorphic function on every type; bar is the identity function on integers; and baz is the identity function on pairs whose second coordinate is boolean. Typically, we want library functions to be highly polymorphic and widely applicable, but the functions defined in EventML programs to be much more constrained. By and large, the polymorphism of an EventML program is expressed in its parameters.

Without variable declarations for ni and sender the definition of the newvote operation would have to be expressed as

```
let newvote (ni:RoundNum) ((ni',c),sender:Loc) (_,locs) = ... ;;
```

but with those declarations, we may simply write

```
let newvote ni ((ni',c),sender) (_,locs) = ... ;;
```

As a practical matter, there's not much point in trying to anticipate where type inference needs hints. Most commonly, help may be needed when the right hand side of the definition calls on a polymorphic function such as deq−member, which operates on lists of any type that has a decidable equality operator.

The balance between introducing variable declarations and adding type labels to patterns and expressions is a matter of taste.

Auxiliaries

We introduce a convenient notation for specifying the "init" parameter of SM∗−class or Memory∗ (section 4):

```
let init x loc = {x} ;;
```

Used in that context, (init x) is the function that assigns the initial state x to every location.

Class combinators

The specification uses one new library combinator:

- X until Y: $v \in$ (X until Y)(e) iff $v \in$ X(e) and no Y-event has previously occurred at loc(e). That is, at any location l, the class (X until Y) acts exactly like X until a Y-event occurs at l, after which it falls silent.

5.1.2 The top level: Replica

Replica is the event class characterizing the actions of a decider. As noted in figure 8, the main program

```
main Replica @ locs
```

installs a decider at each location in locs.

For each n, a Replica will spawn (at most) one instance Voter to communicate with other instances of Voter and come to consensus on a single proposal of the form $(n, \_)$.

```
class Replica = NewVoters >>= Voter ;;
```

Figure 5 2/3 consensus: NewVoters and ReplicaState

```
(* ———————— ReplicaState: a state machine ———————— *)

(* -- inputs -- *)
let vote2prop loc (((n,i),c),loc') = {(n,c)} ;;
class Proposal = propose'base || (vote2prop o vote'base);;

let update_replica (n,c) (max,missing) =
  if n > max
  then (n, missing ++ (from-upto (max + 1) n))
  else if deq-member (op =) n missing
       then (max, list-diff (op =) missing [n])
       else (max,missing) ;;

class ReplicaState = Memory1 (init (0,nil)) update_replica Proposal ;;

(* ———————— NewVoters ———————— *)

let when_new_proposal loc (n,c) (max,missing) =
  if n > max or deq-member (op =) n missing then {(n,c)} else {} ;;

class NewVoters = when_new_proposal o (Proposal, ReplicaState) ;;
```

For each n, NewVoters spawns a Voter in response to the first proposal or vote it receives concerning command n.

We define consensus on proposal $\langle n, c\rangle$ to mean that 2/3 (plus one) of the replicas vote for it. On any particular poll of the voters that degree of consensus cannot be guaranteed—so we allow do-over polls, for which we adopt the following terminology. Successive polls for each command number are assigned consecutive integers called *innings*; the pair $\langle$*command_number*, *inning*$\rangle$ is called the polling or voting *round*.

Votes are of type Vote. Each contains:

- the round in which the vote is cast
- a command being voted for in that round
- the voter's location (used to ensure that repeat votes from the same source are ignored)

5.1.3 ReplicaState and NewVoters

This section refers to figure 5.

A Replica acts when NewVoters does, in response to propose and vote inputs. These are recognized by the class Proposal:

```
let vote2prop loc (((n,i),c),sender) = {(n,c)} ;;
class Proposal = propose'base || (vote2prop o vote'base);;
```

Proposal observes the value of type Proposal input in its input.

ReplicaState maintains the state of a Replica, enough information to recognize the first time it sees a Proposal-event about command n (meaning a value of the form $\langle n, c\rangle$ for some command c). Its state has type Int * (Int List). The Int component is the greatest n for which it has seen such an event; and the (Int List) component is the list of all natural numbers less than that maximum for which it has *not* yet seen a proposal event.

Figure 6 2/3 consensus: Rounds and Quorums

```
(* ---------- QuorumState ---------- *)

let newvote ni ((ni',c),sender) (cmds,locs) =
  ni = ni' & !(deq-member (op =) sender locs);;

let add_to_quorum ni ((ni',c),sender) (cmds,locs) =
  if newvote ni ((ni',c),sender) (cmds,locs)
  then (c.cmds, sender.locs)
  else (cmds,locs);;

class QuorumState ni = Memory1 (init (nil,nil)) (add_to_quorum ni) vote'base ;;

(* ---------- Quorum ---------- *)

let roundout loc (((n,i),c),sender) (cmds,_) =
  if length cmds = 2 * flrs
  then let (k,x) = poss-maj cmdeq (c.cmds) c in
       if k = 2 * flrs + 1
       then bag-append (decided'broadcast locs n)
                       (notify'broadcast clients (n,x))
       else { retry'send loc ((n,i+1), x) }
  else {} ;;

let when_quorum ni loc vote state =
  if newvote ni vote state then roundout loc vt state else {} ;;

class Quorum ni = (when_quorum ni) o (vote'base, QuorumState ni) ;;

(* ---------- Round ---------- *)

class Round (ni,c) = Output(\loc.vote'broadcast locs ((ni,c),loc))
                  || Once(Quorum ni) ;;
```

```
let update_replica (n,c) (max,missing) =
  if n > max
  then (n, missing ++ (from-upto (max + 1) n))
  else if deq-member (op =) n missing
       then (max, list-diff (op =) missing [n])
       else (max,missing) ;;

class ReplicaState = Memory1 (init (0,nil)) update_replica
                             Proposal ;;
```

The initial state of a ReplicaState is $\langle 0, nil \rangle$. The infix operator ++ is the append operator and the imported Nuprl operations from−upto and list −diff have the following meanings:

$$\begin{aligned} \text{from-upto } i\ j &= [i; i+1; i+2; ...; j-1] \\ \text{list -diff (op =)}[a;b;...][m;n;...] &= \text{the result of deleting all occurrences of } m, n, \ldots \text{ from } [a;b;...] \end{aligned}$$

Every event is a ReplicaState-event, and observes the state of the state machine when the event occurs (before any processing).

Figure 7 2/3 consensus: NewRounds and Voters

```
(* ---------- NewRoundsState ---------- *)

let vote2retry loc ((ni,c),sender) = {(ni,c)};;
let RoundInfo = retry'base || (vote2retry o vote'base);;

let update_round n ((m,i),c) round = if n = m & round < i then i else round ;;

class NewRoundsState n = Memory1 (init 0) (update_round n) RoundInfo ;;

(* ---------- NewRounds ---------- *)

let when_new_round n loc ((m,i),c) round =
  if n = m & round < i then {((m,i),c)} else {} ;;

class NewRounds n = (when_new_round n) o (RoundInfo, NewRoundsState n) ;;

(* ---------- Voter ---------- *)

class Halt n = (\_.\m. if m = n then {()} else {}) o decided'base;;

class Voter (n,c) = Round ((n,0),c)
                 || ((NewRounds n >>= Round) until (Halt n));;
```

Figure 8 2/3 consensus: The top level

```
(* ---------- Replica ---------- *)

class Replica = NewVoters >>= Voter;;

(* ---------- Main program ---------- *)

main Replica @ locs ;;
```

NewVoters-events are Proposal-events. NewVoters compares the data observed by Proposal with the state of the replica when the message arrives, in order to decide whether it is the first proposal about some n.

```
let when_new_proposal loc (n,c) (max,missing) =
  if n > max or deq-member (op =) n missing
  then {(n,c)}
  else {} ;;

class NewVoters = when_new_proposal o (Proposal, ReplicaState) ;;
```

5.1.4 The next level: Voter

A Voter is a parallel composition of two classes:

```
class Voter (n,c) = Round ((n,0),c)
                 || ((NewRounds n >>= Round) until Halt n);;
```

where:

- Round$((n,i),c)$ will, at any location, conduct the voting for round (n,i), and will cast its vote in that round for command c.
- NewRounds n >>= Round will determine when it is time to begin a new round of voting for the n^{th} command and spawn a class to conduct the voting in that new round.
- The clause "until Halt n" will cause termination when it detects a Halt n event, which signals that some Voter has found a consensus for command n.

5.1.5 Round and Quorum

This section refers to figure 6.

Round $((n,i),c)$

Round $((n,i),c)$, running at location *loc*, broadcasts a vote from *loc* for command c in round (n,i) and runs an instance of Quorum. Quorum (n,i) keeps a tally of votes received at l in round (n,i) and uses that tally to determine *either* that consensus has been reached (in which case it notifies the clients and sends every Replica, including itself (i.e., the replica that spawned it), a decided message) *or* that consensus might not be possible in inning i (in which case it sends to itself a suitable retry message).

```
class Round (ni,c) = Output(\loc.vote'broadcast locs ((ni,c),loc))
                   || Once(Quorum ni) ;;
```

(Quorum (n,i)) is a state machine that responds to vote messages. Intuitively, its state consists of a pair $\langle cmds, locs\rangle$. Each time it receives a *new* vote for proposal $\langle n,c\rangle$ in inning i, it prepends c to the list *cmds*. *locs* is the list of the locations that sent those commands. (We keep the list of senders so that, if a vote from any sender is delivered multiple times, it will only be counted once.) The initial state is a pair of empty lists. (QuorumState (n,i)) is the "pre" Moore machine that observes this state when a vote arrives.

```
let newvote ni ((ni',c),sender) (cmds,locs) =
  ni = ni' & !(deq-member (op =) sender locs);;

let add_to_quorum ni ((ni',c),sender) (cmds,locs) =
  if newvote ni ((ni',c),sender) (cmds,locs)
  then (c.cmds, sender.locs)
  else (cmds,locs);;

class QuorumState ni = Memory1 (init (nil,nil))
                               (add_to_quorum ni) vote'base ;;
```

The transition function for (QuorumState (n,i)) is (add_to_quorum (n,i)). A vote message is a no-op unless it's a vote in round (n,i) that comes from a new location. If it's both, then the vote is tallied by prepending to it state components the command it votes for and the location of its sender.

Quorum (n,i) is a Mealy machine defined from QuorumState. It produces an output once it has received votes from 2 flrs $+1$ distinct locations. If all of them are votes for the same command d, it broadcasts notify and decided messages. If not, then it is possible that on this round no proposal will ever receive 2 flrs $+1$ votes; so it sends itself a retry message to trigger initiation of inning $i+1$. (Once it has sent the retry message it will ignore any votes it subsequently receives in round $\langle n,i\rangle$, even if they would result in some proposal's receiving 2 flrs $+1$.)

```
let roundout loc (((n,i),c),sender) (cmds,_) =
  if length cmds = 2 * flrs
  then let (k,x) = poss-maj cmdeq (c.cmds) c in
         if k = 2 * flrs + 1
         then bag-append (decided'broadcast locs n)
                         (notify'broadcast clients (n,x))
         else { retry'send loc ((n,i+1), x) }
  else {} ;;

let when_quorum ni loc vote state =
  if newvote ni vote state then roundout loc vt state else {} ;;

class Quorum ni = (when_quorum ni) o (vote'base, QuorumState ni) ;;
```

Consider first the outer conditional. The (cmds,_) argument matches the value observed by QuorumState, so (length cmds) is the number of votes tallied before the input arrives. If this test fails then, even with the new input, the state machine will not yet have received 2 flrs + 1 votes, so the input is ignored.

Consider the inner conditional. The imported operation poss−maj implements the Boyer-Moore majority algorithm. Thus, the locally defined constants k and x have the following meaning: If some element of the list c.comds appears in a majority of its entries, x is that element and k is the number of times it occurs. Thus, the inner conditional tests for unanimity.

The data of a retry message consists of the new round to be initiated and, in addition, the name of a command to propose in this new round. The definition of roundout attempts to choose that command in a reasonable way: So, if the votes are not unanimous, but some command receives a majority, that majority-receiving command will be proposed in the retry message.[8]

5.1.6 NewRounds and Voters

This section refers to figure 7.

Halt n

Halt n recognizes the arrival of decided message with body n. We make it a class of type Unit, since the only information conveyed is the fact that the message has arrived.

```
class Halt n =
    Once((\_.\i.if i = n then {()} else {}) o decided'base) ;;
```

NewRounds n

Recall that (NewRounds n) decides when to initiate a new round of voting about the n^{th} command and, when necessary, spawns an instance of Round, supplying it with a new round number of the form $\langle n, \_\rangle$ and a command to vote for in that round.

(NewRoundsState n) is a "pre" Moore machine. It's state is an integer, initially 0. At any location it keeps track of the greatest inning i for which it has "participated" in a round of the form $\langle n, i\rangle$. A location has "participated" in such a round if it has received a retry message with data $\langle\langle n, i\rangle, \_\rangle$, or a vote message with data $\langle\langle\langle n, i\rangle, \_\rangle, \_\rangle$. So its input events are recognized by RoundInfo, which observes the round/command pair embedded in its input.

```
let vote2retry loc ((ni,c),sender) = {(ni,c)};;
class RoundInfo = retry'base || (vote2retry o vote'base);;
```

[8] This is crucial to the correctness of the protocol.

The transition function, update_round, updates the state whenever its input constitutes participation in an inning greater than the current state value:

```
let update_round n ((m,i),c) round = if n = m & round < i
                                     then i else round ;;
class NewRoundsState n =
    Memory1 (init 0) (update_round n) RoundInfo ;;
```

There's some redundancy in defining the Mealy machine NewRounds from NewRoundsState.[9] The condition in when_new_round is the same as that in the transition function updated_round: when the transition is a no-op, NewRounds ignores the input; when it's not, NewRounds passes along the input that caused the update.

```
let when_new_round n loc ((m,i),c) round =
  if n = m & round < i then {((m,i),c)} else {} ;;
class NewRounds n =
    (when_new_round n) o (RoundInfo, NewRoundsState n) ;;
```

5.2 Illustrative runs of the protocol

This section contains message sequence charts that describe some possible runs of the 2/3-consensus protocol. To make the charts easier to read, all message arrows are drawn horizontally (except for self-messages).[10] That requires a small, but semantically inessential, deviation from the official semantics of EventML. Actions that are atomic in EventML may be shown as nonatomic. Consider figure 9. The top diagram shows A broadcasting message x to B, C, and D as a single event. At C, the act of receiving message x and replying with y is atomic. The second diagram teases everything apart.

We can represent delay in message transit, in part, as a delay in sending the message. Since only message arrivals are observable, no distinction between the picture and the official semantics will be observable.

A detailed look at retry Figure 10 shows (part of) one possible run of the consensus protocol, in which a round ends not in consensus but in a retry that starts a new round. We assume that flrs $= 1$, so there are four instances of Replica and a proposal will be accepted if it gets three votes. The diagram does not depict all the classes—in particular, we show only three of the replicas—and does not display all the messages sent. It contains abbreviations, which are defined in the following table:

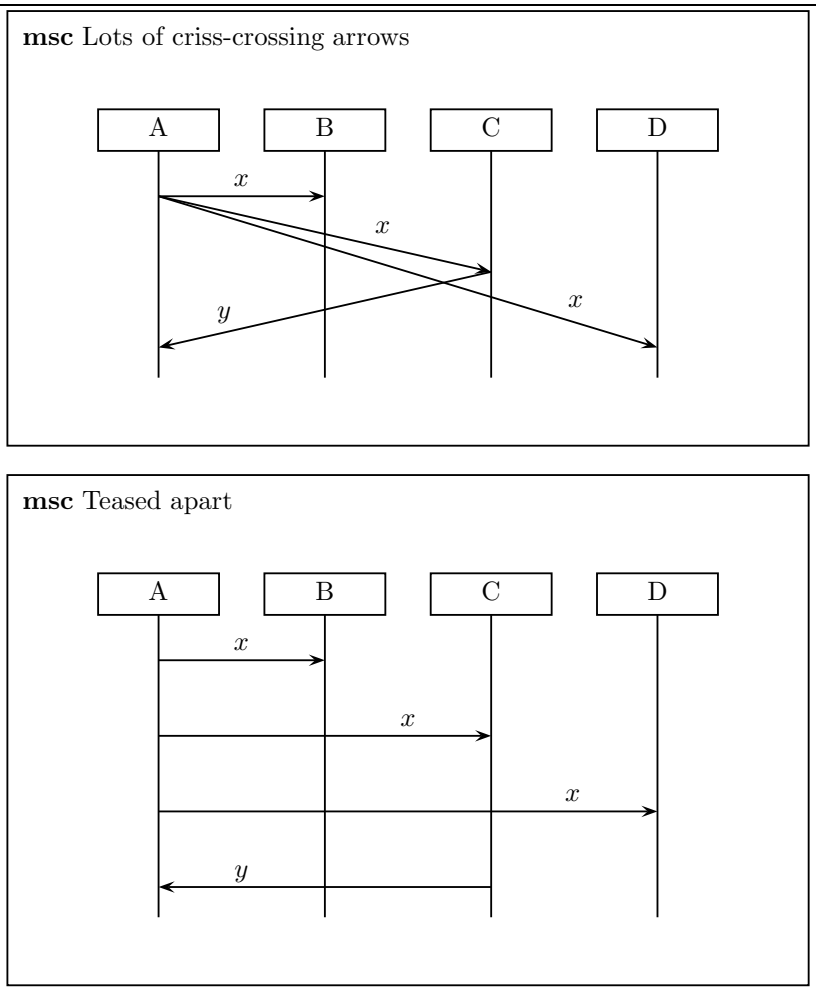
$vote_{1x}$	=	[vote : $((2,0),x,l_1)$]
$vote_2$	=	[vote : $((2,0),x,l_2)$]
$vote_{4y}$	=	[vote : $((2,0),y,l_4)$]
$retry_x$	=	[retry : $((2,1),x)$]
$vote'_{2x}$	=	[vote : $((2,1),x,l_2)$]
α	:	start Round ((2,0),x); Quorum state = $([x],[l_1])$
β	:	Quorum state = $([x;x],[l_1;l_2])$
γ	:	Quorum state = $([x;x;y],[l_1;l_2;l_4])$
δ	:	start Round ((2,1),x); Quorum state = $([x],[l_2])$

Note that votes not marked with a "′" are cast in inning 0 (i.e., in this case, round (2,0)) and votes marked with "′" are cast in inning 1.

This run begins when the Replica at location l_1 receives a proposal $(2, x)$ from the environment. We assume that location l_1 has not previously received a vote or proposal for command 2; accordingly, it

[9] The next version of the library will contain a different set of combinators that avoids that.
[10] A horizontal arrow does *not* imply instantaneous communication.

Figure 9 Simplifying pictures



responds by spawning an instance of Voter $(2, x)$ at l_1. Only one component of this Voter will play a role: Round $((2, 0), x)$. This class broadcasts $vote_{1x}$, a vote for the proposal it received—though the diagram shows only two of those messages. Its Quorum component plays no role in this part of the run.

A Replica can respond to either a vote or a proposal. When the Replica at location l_2 receives $vote_{1x}$ (also assumed to be new), it spawns an instance of Voter $(2, x)$ at l_2. This initiates an instance of Round $((2, 0), x)$ at location l_2, which will broadcast $vote_2$ and spawn an instance of Quorum $(2, 0)$ at l_2. Of this broadcast we show only the message it sends to itself.[11] Comment α says that the vote that spawned the Round updates the internal state of Quorum to $([x], [l_1])$, recording the fact that a vote for command x came from l_1. As β indicates, the self message updates the state of this Quorum to $([x; x], [l_1; l_2])$.

Meanwhile, the Replica at location l_4 has received a competing proposal: that command 2 be y, not x. It spawns Voter $(2, y)$, which broadcasts $vote_{4y}$; we show only the message received by the Voter at l_2. This updates the state of Quorum at l_2 to $([x; x; y], [l_1; l_2; l_4])$. Once it has received votes from three distinct locations Quorum makes a decision: in this case, because the votes are not unanimous, it must start a new round by sending itself a *retry* message.[12] As δ indicates, this retry starts Round $((2, 1), x)$. So the Voter at l_2 begins by broadcasing $vote'_{2x}$.

Notification and retry in the same round Figure 11 shows part of a run in which the Voter at l_1 broadcasts a notification that the second command will be x, but the Voter at l_2 sends a retry that launches a new round. As before, the diagram does not depict all the classes or all the messages sent. Instead of walking through the successive states of the Quorum classes, we only note their states when they reach a decision. The abbreviations are as follows:

$vote_{1x}$	=	[vote : $((2, 0), x, l_1)$]
$vote_{2x}$	=	[vote : $((2, 0), x, l_2)$]
$vote_{3x}$	=	[vote : $((2, 0), x, l_3)$]
$vote_{4y}$	=	[vote : $((2, 0), y, l_4)$]
$decided_x$	=	[decided : $(2, x)$)]
$notify_x$	=	[notify : $(2, x)$] is broadcast to all clients
$retry_x$	=	[retry : $((2, 1), x)$]
α	:	Quorum state = $([x; x; x], [l_1; l_2; l_3])$
β	:	Quorum state = $([x; x; y], [l_1; l_2; l_3])$
γ	:	start Round ((2,1),x); Quorum state = $([x], [l_2])$

The first three votes seen by the Voter at location l_1 are votes for x, so it notifies all clients that agreement has been reached—command 2 is x—and sends a decided message to stop all the Voters working on command 2. The Voter at location l_2 sees two votes for x and one for y and it launches a new round before it receives the decided message that stops it. The crucial point is that, on launching this round it casts its vote for x. If the retry proposed y, it might be possible that the remaining voters in some later round would come to consensus on command y; clients would then receive a contradictory notification saying that command 2 is y. Section 5.3 explains why this calamity cannot occur.

Failure to achieve consensus Figure 12 illustrates a run in which this protocol fails to achieve consensus, a possibility that, according to the FLP theorem [FLP85] is inevitable. The abbreviations are as follows:

[11] The Replica at location l_2 sees this vote but, as it has already seen a vote for command 2, the self message does not cause it to spawn a new Voter.

[12] It is possible that the fourth Replica would cast a vote for proposal $(2, x)$, providing the three votes, but that would come too late.

Figure 10 Detailed example of a retry

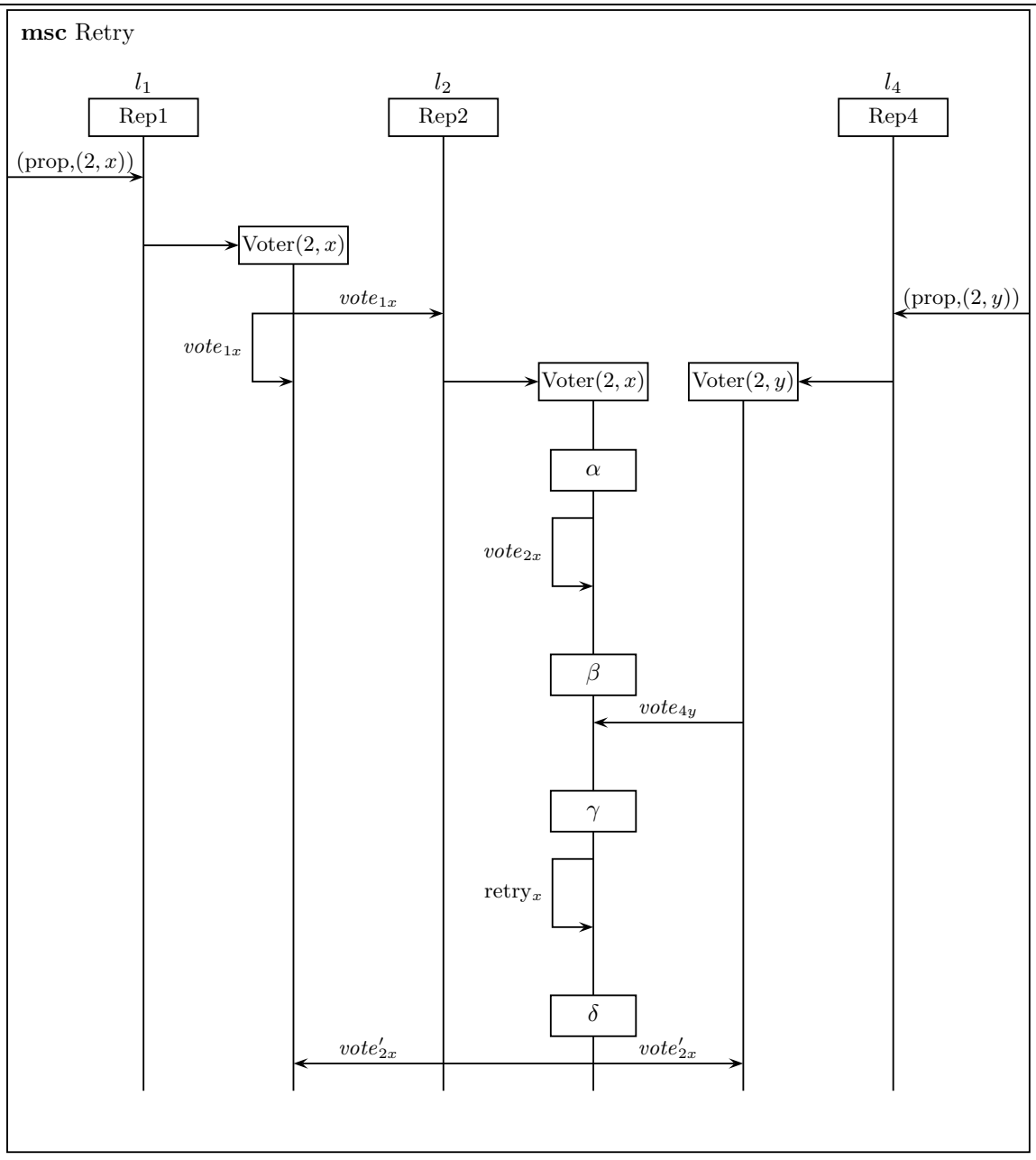


Figure 11 Notify and Retry on the same round

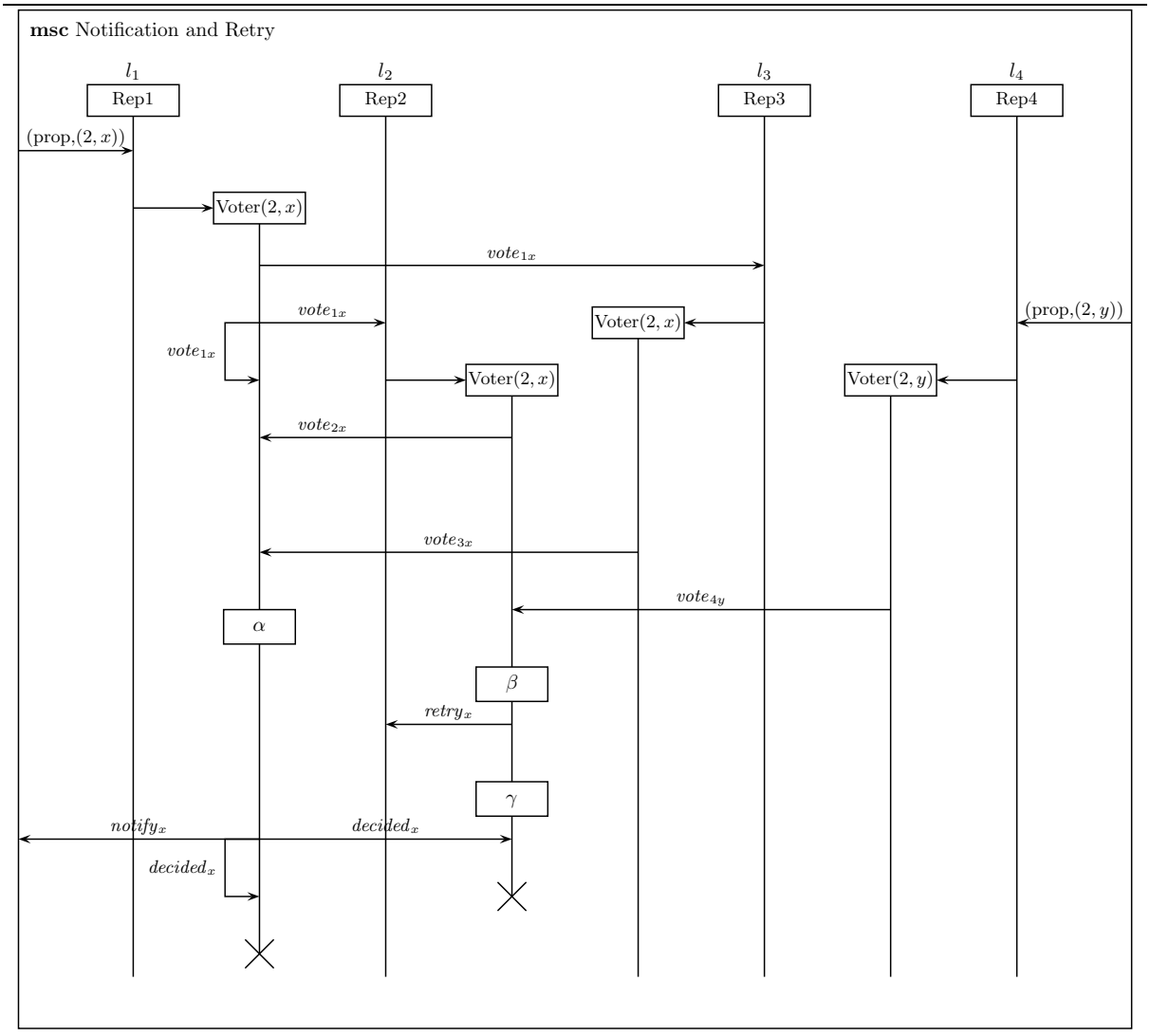
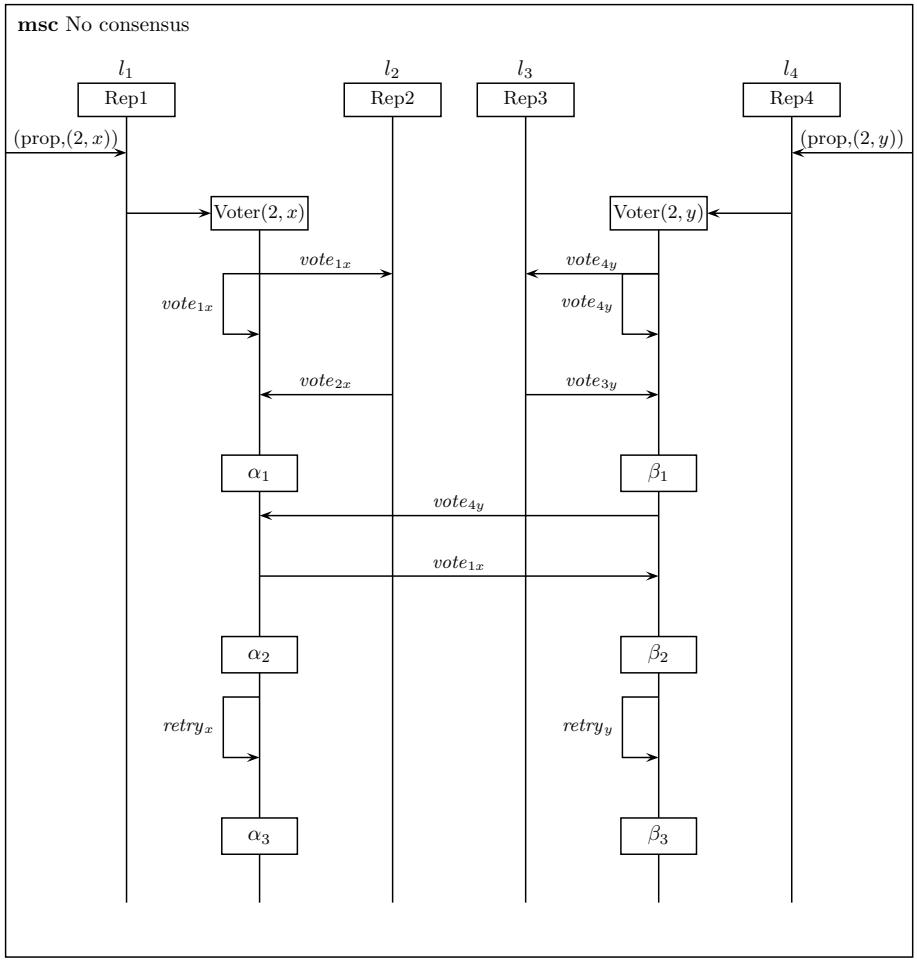


Figure 12 Failure to achieve consensus



$vote_{1x}$	=	$[\mathsf{vote} : ((2,0), x, l_1])$
$vote_{2x}$	=	$[\mathsf{vote} : ((2,0), x, l_2])$
$vote_{3y}$	=	$[\mathsf{vote} : ((2,0), y, l_3])$
$vote_{4y}$	=	$[\mathsf{vote} : ((2,0), y, l_4])$
$retry_x$	=	$[\mathsf{retry} : ((2,1), x])$
$retry_y$	=	$[\mathsf{retry} : ((2,1), y])$
α_1	:	Quorum state $= ([x;x],[l_1;l_2])$
α_2	:	Quorum state $= ([x;x;y],[l_1;l_2;l_4])$
α_3	:	start Round ((2,1),x); Quorum state $= ([x],[l_1])$
β_1	:	Quorum state $= ([y;y],[l_4;l_3])$
β_2	:	Quorum state $= ([y;y;x],[l_4;l_3;l_1])$
β_3	:	start Round ((2,1),y); Quorum state $= ([y],[l_4])$

We omit the Voter classes spawned at locations l_2 and l_3, depicting their messages as coming directly from the Replica classes themselves. In round (2,0), the Replica at l_1 votes for x and the Replica voting for y. This exchange of messages results in abandoning round (2,0). But round (2,1) begins in exactly the same way: with Replica at l_1 voting for x and at l_2 voting for y. The pattern can in principle repeat endlessly.

5.3 Properties of the 2/3-consensus protocol

Consistency The 2/3-consensus protocol must satisfy the basic safety property of *consistency*—if the messages ([notify : $\langle n, x\rangle$]) and ([notify : $\langle n, y\rangle$]) are sent, then $x = y$. The example in figure 11 makes it clear that the following property is necessary (though not quite sufficient) to guarantee that.

> If, in any round, some Voter finds a quorum for command x, then, in that round, x is the only command that can be proposed by a retry message.

PROOF: Suppose that one Voter sees $2f + 1$ votes for command x in a given round. Since each Voter votes for only one command in any round, that round can contain no more than f votes for any command other than x. Now consider the situation of any other Voter making a decision in that round: It will have received $2f + 1$ votes, and at most f of them can be for a command other than x. Therefore, at least $f + 1$ of the votes it sees must be for x; so if it sends a retry message, that retry proposes x.

The argument is not quite done. Suppose one Voter finds a quorum for x in round $\langle n, i\rangle$ but other Voters do not, and will therefore participate in subsequent rounds. Is it possible that one of those later rounds contains a vote for some other command y (possibly as the result of a new proposal received from some external source), and that, as a result, some later round $\langle n, j\rangle$ finds a quorum for y? No, because a stronger property holds.

> If some Voter finds a quorum for command x in round $\langle n, i\rangle$ then in any round $\langle n, j\rangle$ with $j >$ i all votes cast are votes for x.

PROOF: Every vote can ultimately be traced either to a retry message or to a proposal message received by some Replica from an external source. However, a Replica will ignore a proposal with body $\langle n, c\rangle$ unless it has *never* before received either a proposal or a vote for something of the form $\langle n, \_\rangle$. Thus, votes that arise from external proposals can be cast only in rounds of the form $(\_, 0)$. That is to say that all votes in round $\langle n, i\rangle$ with $i > 0$ arise from retry messages sent in round $\langle n, i - 1\rangle$. So, by induction, once we encounter any round in which all retry messages are for command x, all subsequent rounds can only contain votes for x.

Fault tolerance When a process suffers a *crash failure* it stops sending messages. (It does not perform erratically by, e.g., violating the requirements of the protocol.) The 2/3-consensus protocol will tolerate up to flr crash failures, in the following sense:

> All executions of the protocol that suffer only crash failures, and no more than flr of those, are *non-blocking*—that is, execution never reaches a state from which consensus is impossible.

By the FLP theorem, this is the strongest fault tolerance guarantee that a consistent consensus protocol can provide.

6 Paxos

Paxos is also a consensus protocol for coordinating the behavior of copies of a state machine that has been replicated to tolerate crash failures. It requires only $2n + 1$ copies in order to tolerate n failures. The EventML program formalizes the pseudo-code description of Paxos given in [Ren11]. We assume that the reader is familiar with that paper and has a copy in hand.

6.1 A gross description of the protocol

This section gives a very high-level description of how participants in the protocol—Replicas, Leaders, and Acceptors—interact. It omits much of the combinatorial detail that makes the protocol work.

6.1.1 Replicas

A Replica puts a wrapper around a copy of the state machine we want to replicate. To invoke an operation on the state machine a client sends every Replica a request that contains a Command consisting of three things: the client's return address; the name of an operation to be performed; an identifier, a "client id," chosen by the client. After the operation is performed the client will receive a message containing both the result and that identifier—so clients should give distinct requests distinct client id's, but the correctness of the protocol does not depend on that.

On receiving requests, Replicas formulate proposals. A Proposal is a pair consisting of a positive integer and a Command.[13] Intuitively, $\langle n, cmd \rangle$ "proposes" that cmd should be the n^{th} command performed by each copy of the state machine; we call n the *slot number* of the proposal.

Replicas send proposals to Leaders; Leaders send to Replicas decision messages containing the Proposal values on which the protocol has achieved consensus.

More specifically, when a Replica receives a request for command cmd, it chooses the least slot number n for which it has neither made a previous proposal nor received a decision and sends the proposal $\langle n, cmd \rangle$ to every Leader. A Replica that has proposed $\langle n, cmd \rangle$ may subsequently receive a decision message choosing some other command for slot n; if so, and if the Replica has not already executed cmd, the Replica will propose the command for a different slot. (Note: Leaders may receive proposals to put the same command in different slots and, as a result, the same command may be chosen for multiple slots; but it will not be executed more than once.)

There may be gaps in what a Replica knows—e.g., it may have received $\langle 1, cmd_1 \rangle$, $\langle 2, cmd_2 \rangle$, and $\langle 4, cmd_4 \rangle$, in decision messages, but no such message about any proposal of the form $\langle 3, \_\_ \rangle$. In this state, a Replica may execute cmd_1 and cmd_2, in that order[14], but may not execute any further commands until it has received a decision about the command for slot number 3. The details of the bookkeeping make Replicas the most algorithmically complex of the participants, but they are from our point of view the least interesting, because the actions of the Replicas are essentially independent of the protocol used to obtain consensus.

6.1.2 Leaders and Acceptors

The heart of the protocol is the interaction between Leaders and Acceptors, which can be defined independently of the beavior of the Replicas. Leaders engage in a dialogue with the Acceptors to determine which of the proposals they receive will be agreed to. Acceptors vote and their votes are tallied by processes that the Leaders spawn. As usual, any one election need not yield a decision; so re-votes must be accommodated. Each Leader has an unbounded supply of ballot numbers (of type Ballot_Num), disjoint from the supply of every other Leader. The data contained in a vote is a "pvalue" consisting of a ballot number and a Proposal (voted for on that ballot). The set of *all* ballot numbers is linearly ordered.

The Leader/Acceptor dialogue consists of two phases, repeated as often as necessary. To carry out each instance of phase 1 a Leader chooses a ballot number from its supply and spawns a Scout process that exchanges messages with Acceptors to determine whether that ballot number will be *adopted* or *preempted*.[15] Scouts send p1a messages and Acceptors reply with p1b messages. If the ballot number is preempted, the Leader will choose a new ballot number and spawn a new Scout for it. (As will be seen, a ballot number may also be preempted in phase 2.) If its ballot number is adopted, a Leader proceeds to phase 2.

To carry out phase 2, a Leader spawns Commander processes that solicit votes from Acceptors. Each Commander is associated with a pvalue, $\langle b, \langle s, p \rangle \rangle$, where b is a ballot number adopted in phase 1 and not (yet) preempted. (Call this an "active" ballot number.) The Commander asks the Acceptors for

[13] Depending on context, [Ren11] uses the word "proposal" sometimes for a value of type Command and sometimes for one of type Proposal.

[14] Assuming that neither of them has previously been executed.

[15] These messages will contain additional data needed for bookkeeping.

the highest ballot number they've seen (including the b they've just received). Commanders send p2a messages and Acceptors reply with p2b messages. The Commander for $\langle b, \langle s, p\rangle\rangle$ tallies the replies from Acceptors, which will result either in choosing $\langle s, p\rangle$ (in which case the Commander notifies the Replicas with decision messages) or preempting b (causing the Leader to choose a new ballot number and spawn a new Scout).

The phase 1 and phase 2 voting dialogues are described further in section 6.1.3.

All a Leader does is spawn Scouts and Commanders. Its state consists of three things: a ballot number; a boolean that is true iff that ballot number is active; a list of proposals that the Leader is working on. This list will always be consistent (as defined in section 6.3). A Leader begins by spawning a Scout for the least ballot number that it owns and thereafter responds to preempted, propose, and adopted messages.

A preempted message, which may come from a Scout or a Commander, causes a Leader to choose a new ballot number (which will be "bad" unless and until it becomes adopted) and spawn a Scout for it.

A propose message comes only from a Replica. In response to [propose : $\langle s, p\rangle$], a Leader acts as follows: If it has already received a proposal for slot s it ignores the input. If not, it adds $\langle s, p\rangle$ to the list of proposals it's working on; and if, in addition, its current ballot number b is active, it spawns a Commander for $\langle b, \langle s, p\rangle\rangle$.

If a Leader receives an adopted message for its current ballot number (call it bn), the Leader first uses additional data in the message to update its list of "proposals I'm currently working on." (The details are somewhat complex.) Then, for each proposal $\langle s, p\rangle$ in that revised list it spawns a Commander for $\langle bn, \langle s, p\rangle\rangle$. (An adopted message for a ballot number other than bn will be ignored.)

6.1.3 The voting

Phase 1, simplified: A Scout for ballot number b sends every Acceptor a p1a message containing b. An Acceptor replies with a p1b message containing (among other things) the greatest ballot number that it has so far received (in either a p1a or p2a message)—which is guaranteed to be a ballot number at least as great as b. The Scout tallies responses and comes to a decision as soon as one of the following two things happens:

1. It receives a p1b message with a value $b' \neq b$.

   In this case, the Scout sends the Leader a preempted message with value b'. The Leader will update its current ballot to a value greater than b' and spawn a new Scout for it.

2. It receives p1b messages with value b from a majority of the Acceptors.

   In this case, the Scout sends the Leader an adopted message with value b. What makes this a simplified account: an Acceptor sends extra information to a Scout in its p1b message and a Scout sends extra information to its Leader in its adopted message. Explaining that will require some discussion of phase 2.

Because of (1), the Scout will declare its ballot number adopted only if it receives a *unanimous* majority. If, before that occurs, it receives a single reply with a value other than b then it is logically possible that, because of failures, a majority for b cannot exist.

Phase 2, simplified: A Commander for the pvalue $\langle b, \langle s, p\rangle\rangle$ sends a [p2a : $\langle b, \langle s, p\rangle\rangle$] message to every Acceptor. Each Acceptor responds with a p2b message containing the largest ballot number it has seen (including b). The Commander tallies these responses just as a Scout does: If it receives a p2b message with a $b' \neq b$ before it receives a unanimous majority for b it preempts; otherwise, it broadcasts a decision message for proposal $\langle s, p\rangle$.

Figure 13 Paxos parameters

```
parameter accpts : Loc Bag  (* Locations of acceptors *)
parameter reps   : Loc Bag  (* Locations of replicas  *)
parameter ldrs   : Loc Bag  (* Locations of leaders   *)
(* We're assuming that none of these bags has repetitions. *)

parameter ldrs_uid : Loc → Int
(* This function is used to order the leader locations. *)


parameter Op,  eqOp   : Type * Op  Deq (* Operations invoked by replicas.  *)
parameter Cid, eq_Cid : Type * Cid Deq (* Command identifiers              *)

parameter Result      : Type           (* Results returned by replicas    *)
parameter RepState    : Type           (* State of SMs to be replicated   *)

parameter init_rstate : RepState       (* Common initial state of SMs     *)
parameter apply_op    : Op → RepState → RepState * Result
```

Phases 1 and 2, unsimplified: If an Acceptor responds to a [p2a : $\langle b, \langle s, p\rangle\rangle$] message from a Commander by returning b, we say that it *accepts* that pvalue. An Acceptor keeps track of all the pvalues it has thus far accepted and includes that set in the p1b messages it sends to Scouts. When a Scout finds that a majority has accepted its ballot number, it includes in the adopted message it sends to the Leader a set consisting of all the pvalues accepted by all the Acceptors that accepted it. The Leader will use this information to constrain the way it spawns Commanders, so as to guarantee consistency in decisions: if there are decisions for slot n on multiple ballots they all choose the same command.

Following these rules literally means that the state of an Acceptor and the contents of p1b and adopted messages can grow without bound. [Ren11] notes simple optimizations that achieve the desired result but keep the sizes of the states and messages bounded.

6.2 Parameters

Figure 13 declares the parameters of the specification. This section explains their intended meanings.

Acceptors will reside at locations in the bag accpts, Replicas at locations in reps, and Leaders at locations ldrs. We will assume that none of these bags contains repetitions but do not assume that they are disjoint.

The main program, given in figure 24, is

```
main Leader @ ldrs || Acceptor @ accpts || Replica @ reps
```

Instances of event classes implementing the Scout and Commander will be spawned by, and therefore co-located with, Leader classes.

We use the function parameter ldrs_uid to order the locations of leaders, and thereby to order the ballot numbers; we therefore assume that it maps each element of ldrs to a different integer.

Cid is the type of client id's. Recall: A user who sends the replicas a request to perform some operation will tag the request with a value from Cid; the return value that the user receives will be tagged with the same value. Clients should tag their requests sensibly, but the protocol doesn't care how they do it.

The remaining parameters model the abstract state machine being replicated. It has a state, of type RepState, whose initial value is init_state . Inputs to the state machine, of type Op, name operations to be performed. The transition function apply_op takes an Op and RepState as arguments and returns a pair consisting of a new RepState and an output value of type Result. Our current notion of configuration

Figure 14 Paxos types and variables

```
type Ballot_Num = (Int * Loc) + Unit;;
type Command    = Loc * Cid * Op ;;
type Slot_Num   = Int ;;
type Proposal   = Slot_Num * Command;;
type PValue     = Ballot_Num * Proposal ;;
type PVlist     = PValue List;;
type ReplicaState = RepState * Slot_Num * Proposal List * Proposal List ;;

(* ------- VARIABLES ------- *)

variable loc, loc1, loc2  : Loc
variable cid, cid1, cid2  : Cid
variable op1, op2 : Op
variable slt, slt1, slt2  : Slot_Num
variable bnum, bnum1, bnum2  : Ballot_Num
variable cmd : Command
variable state : ReplicaState
```

file (section 8) is not rich enough to specify how to map these parameters to the actions of a state machine we wish to replicate.

6.3 Types and variables

The types defined in figure 14 encode the types used in [Ren11]. This section a few details of the encoding that may not be immediately obvious.

As noted, the ballot numbers are ordered. They are of two kinds: a "normal" ballot number is a pair containing an integer and the location of a Leader; the "special" ballot number $\perp$ is less than every normal one. The type Ballot_Num is therefore a disjoint union: the normal values belong to the left hand side of the union and $\perp$ is the unique element of the right hand side, represented by the value dummy_ballot = inr (). (This constant is declared later in the specification, along with the constants used to represent initial values.)

Replicas use values of type Slot_Num to index commands in the order in which they are to be performed. These index values will always be positive integers. Although the type of positive integers is definable in Nuprl, it is not definable in EventML. So we make Slot_Num a synonym for Int.

Type ReplicaState represents values of the internal state of the process *Replica* defined in [Ren11], the wrapper process that includes the state machines we want to replicate. The state variables *proposals* and *decisions* that [Ren11] represents as sets of proposals we represent as lists of proposals; EventML does not provide a type constructor for sets. (For technical reasons, explained in section 6.12, the internal state of the event class Replica will contain one additional component.)

It will be an invariant that each of these lists is *consistent* in the following sense: A list of pairs is consistent if, whenever it contains both $\langle x, y\rangle$ and $\langle x, y'\rangle$, then $y = y'$. It is possible in Nuprl to define a type consisting of consistent lists, but such a type definition is not possible in EventML.

6.4 Imports

The SM*−class operations and Memory3 are class combinators defined in section 4. The operations map, deq−member, bag−size, and bag−remove have already been encountered in section 5.

- quicksort−int is what the name says: It sorts integer lists in ascending order via quicksort.
- If $P : T \to$ Bool and $L :$ `List`(T). Then

Figure 15 Paxos imports

```
import map mapfilter deq-member
       bag-append bl-exists bag-size bag-remove
       SM1-class SM2-class Memory3
```

- (filter P L) is the sublist of L consisting of elements that satisfy P.
- If, in addition, $f : T \to T'$, then mapfilter f P = map f (filter P L)
- (bl−exists L P) returns a boolean; its value is true iff some element of L satisfies P.

- If $f : S \to T \to S$ and $y : S$ and $x_i : T$ then list_accum f y $[x_1; x_2; x_3] = f\ (f\ (f\ y\ x_1)\ x_2)\ x_3$

6.5 Auxiliary functions

Figure 16 defines operations that perform a number of auxiliary bookkeeping duties. The reader should consult it, and the explanations below, as needed.

6.5.1 Equality tests

Because their definitions make use of declared variables, the operations same_command, same_proposal, and same_pvalue are equality tests specifically for the types Command, Proposal, and PValue.

6.5.2 Operations on lists

As noted, we represent sets of proposals as lists of proposals. Each proposal is a pair consisting of a slot number and a command. We need a few operations for querying and manipulating them.

$$\text{in\_range}\ \ deq\ z\ [(x_1, y_1); (x_2, y_2); ...]$$

is a boolean value that is true iff z is one of the y_i. The argument deq denotes an operation that decides equality among the y's.

We also introduce two operations for adding elements to lists without introducing repetitions.

$$\text{add\_if\_new}\ eq\ [x_1; x_2; \ldots; x_n]\ v = \begin{cases} [x_1; x_2; \ldots; x_n] & \text{if } v \text{ is one of the } x_i \\ [x_1; x_2; \ldots; x_n; v] & \text{if not} \end{cases}$$

The operator eq must be an appropriate equality decider. That requirement is enforced by typechecking, because the definition of add_if_new contains the expression (eqof eq). The operation eqof is predefined in EventML. If eq is an equality decider, (eqof eq) is eq; if not, (eqof eq) is ill-typed.

If eq is an equality decider, then

$$\text{append\_new\_elems}\ eq\ [x_1; \ldots; x_n]\ [y_1; \ldots; y_m] = [x_1; x_2; \ldots; x_n; a_1; \ldots; a_k]$$

where the list of a's contains one copy of each y_i that is not an x_j.

6.5.3 Operations on ballot numbers

The basic operations on ballot numbers are as follows:

- leq_bnum is the less-then-or-equal-to relation.

  leq_bnum' is an auxiliary used only to define leq_bnum. As in [Ren11], normal ballot numbers are ordered lexicographically; we use the function parameter ldrs_uid to order leaders.

Figure 16 Paxos part 1, auxiliaries

```
let same_command  (loc1,cid1,op1) (loc2,cid2,op2) = loc1 = loc2 & cid1 = cid2 ;;
let same_proposal (slt1,cmd1) (slt2,cmd2) = slt1 = slt2 & same_command cmd1 cmd2;;
let same_pvalue   (bnum1,prp1) (bnum2,prp2) = bnum1 = bnum2 & same_proposal prp1 prp2 ;;

let in_domain deq x xys = deq-member deq x (map fst xys) ;;

let add_if_new test val lst =
  if bl-exists lst (test val)
  then lst
  else lst ++ [val] ;;

let append_news test = list_accum (\a.\x.add_if_new test x a);;

let leq_bnum' (i1,l1) (i2,l2) = i1 < i2 or (i1 = i2 & ldrs_uid l1 <= ldrs_uid l2);;

(* leq : Ballot_Num → Ballot_Num → Bool ;;*)
let leq_bnum bn1 bn2 = !(isl bn1) or (isl bn1 & isl bn2 & leq_bnum' (outl bn1) (outl bn2));;

let lt_bnum' (i1,l1) (i2,l2) = i1 < i2 or (i1 = i2 & ldrs_uid l1 < ldrs_uid l2);;

let lt_bnum bn1 bn2 =
  (!(isl bn1) & isl bn2)
  or
  (isl bn1 & isl bn2 & lt_bnum' (outl bn1) (outl bn2));;

(* max : Ballot_Num → Ballot_Num → Ballot_Num ;;*)
(* If they're equal, we take bn2. *)
let max_bnum bn1 bn2 = if leq_bnum bn1 bn2 then bn2 else bn1;;

let pmax pvals =
  (* We keep only the ones where the slot_number is = and the ballot num is > *)
  let g bn slt (bn',(s',_)) = s = s' & lt_bnum bn bn' in
  (* P says that bn has to be a isl. *)
  let P (bn, (s, c)) = !(bl-exists pvals (g bn s)) in
    mapfilter snd P pvals;;

let update_proposals proposals1 proposals2 =
  list_accum (\a.\ (slt,p).
                  if bl-exists proposals2 (\ (s',_). slt = s')
                  then a
                  else (slt,p) . a)
             proposals2
             proposals1 ;;

(* A computed parameter *)
let threshold = (bag-size accpts + 1) / 2 ;;
```

- lt_bnum is the strict-less-then relation (defined from the auxiliary lt_bnum').
- max_bnum bn_1 bn_2 returns the maximum of $\{bn_1, bn_2\}$.

6.5.4 Auxiliaries introduced in [Ren11]

pmax : PVlist → PVlist implements the operation *pmax*.

From a list *pvs* of pvalues, it produces a consistent list of proposals by giving preference to proposals voted for in later ballots. More precisely, we'll say that the proposal $\langle n, c\rangle$ is *maximal* in the list *pvs* if for some ballot number b, $\langle b, \langle n, c\rangle\rangle \in pvs$ and for no ballot number $b' > b$ does *pvs* contain an element of the form $\langle b', \langle n, \_\_\rangle\rangle$. Then pmax(*pvs*) consists of all the maximal elements of *pvs*. This will be a consistent list of proposals provided that *pvs* is a consistent list of pvalues—that is, *pvs* does not contain two entries having the same ballot number but different proposals. We will apply pmax only to consistent lists of pvalues.

update_proposals : PVlist → PVlist → PVlist implements the operator ⊕.

This is essentially the override operator for partial functions. If *xs* and *ys* are consistent lists of proposals, then (update_proposals *xs ys*) is consistent: it contains all the proposals in either list, except that, when there is a conflict, the conflicting proposal from *xs* is omitted.

6.5.5 Iterating a Mealy machine

Consider the Mealy machine with input type I, state type S, output type Bag(R), and transition function $tr : I \to S \to S \times \texttt{Bag}(R)$

We use iterate_tr to compute the result of applying this state machine to a sequence of inputs. That is, if $ops : \texttt{List}(I)$ is the list of inputs to be processed and and $init : S$ is the initial state then (iterate_tr *tr init ops*) is a pair $\langle s, rs\rangle \in S \times \texttt{Bag}(R)$ such that s is the final state after consuming all the inputs in *ops*, in order, and *rs* is the bag containing *all* the outputs produced along the way.

6.5.6 Class combinator: OnLoc

The polymorphic combinator OnLoc is primitive. For any type T and any function $F : \text{Loc} \to \texttt{Class}(T)$, (OnLoc F) : Class(T) is the event class that, at any location l, acts like the class (F l).

It is defined in the underlying computation model by $\text{OnLoc}(F) = \lambda es.\lambda e.(F\ \text{loc}(e)\ es\ e)$

6.6 Interface

Figure 17 declares all the messages used in [Ren11]. The comments indicate who sends which kind of message to whom. (Recall that each Scout and Commander is co-located with a Leader; so a message sent to or from one of them is sent to or from the location of its Leader.)

6.7 Initial values

Figure 18 defines the initial values for various state machine classes. Note that the function init_leader assigns initial values to instances of Leader based on their locations: the initial ballot number for each Leader is the least ballot number that it owns.

6.8 Acceptors

An Acceptor acts like a state machine. Its input events are the arrivals of p1a or p2a messages and its outputs are p1b and p2b messages. Its state has type Ballot_Num ∗ (PVlist), and is initially (dummy_ballot, nil).

Figure 17 Paxos part 1, interface

```
input  request  : Command       (* client    → Replica *)
output response : Cid * Result  (* Replica   → client  *)

internal p1a : Loc * Ballot_Num
 (* Scout → Acceptor, The Loc is the Scout's leader, *)
internal p1b: Loc * (Ballot_Num * PVlist)
(* Acceptor → Scout, The Loc is the Acceptor's location *)
internal p2a: Loc * PValue
 (* Commander →  Acceptor, The Loc is the Commander's leader, *)
internal p2b: Loc * Ballot_Num
  (* Acceptor → Commander,  The Loc is the Acceptor's location *)
internal preempted: Ballot_Num
 (* Commander, Scout →  Leader *)
internal adopted: Ballot_Num *  PVlist
 (* Scout → Leader *)
internal propose: Proposal
  (* Replica → Leader *)
internal decision: Proposal
  (* Commander → Replica *)
```

Figure 18 Paxos initial values

```
let dummy_ballot : Ballot_Num = inr () ;;

let init_accepted : PVlist = [] ;;
let init_acceptor = (dummy_ballot, init_accepted) ;;

let init_slot_num : Slot_Num = 1 ;;
let init_proposals : Proposal List = [] ;;

let init_pvalues : PVlist = [];;
let init_scout = (accpts, init_pvalues);;

let init_ballot_num loc : Ballot_Num = inl (0,loc);;
let init_active = false ;;
let init_leader loc = (init_ballot_num loc, init_active, init_proposals) ;;

let init_decisions : Proposal List = [] ;;
let init_latest_decision : Proposal + Unit = inr() ;;
let init_replica : ReplicaState = (init_rstate,
                                   init_slot_num,
                                   init_proposals,
                                   init_decisions) ;;
```

Figure 19 Acceptor

```
let on_p1a loc (_,x) (ballot_num, accepted) =
  (max_bnum x ballot_num, accepted);;

let on_p2a loc (_,(b,sp):PValue) (ballot_num, accepted) =
  let ballot_num' = max_bnum b ballot_num in
  let accepted' = if leq_bnum ballot_num b
                  then add_if_new same_pvalue (b,sp) accepted
                  else accepted in
    (ballot_num', accepted') ;;

class AcceptorState =
  SM2-class (\l.{init_acceptor})
            (on_p1a, p1a'base)
            (on_p2a, p2a'base) ;;

class AcceptorsP1a =
  let f loc (ldr,_) bnum_acc = {p1b'send ldr (loc,bnum_acc)}
  in f o (p1a'base, AcceptorState) ;;

class AcceptorsP2a =
  let f loc (ldr,_) (bnum, _) = {p2b'send ldr (loc,bnum)}
  in f o (p2a'base, AcceptorState) ;;

let Acceptor = AcceptorsP1a || AcceptorsP2a ;;
```

We first define the "post" Moore machine AcceptorState: $s \in \mathsf{AcceptorState}(e)$ iff s is the value of the state after processing input event e. We then define Acceptor in terms of it, using the simple composition combinator to compute the output(s) from the input and the state.

Since AcceptorState-events are the p1a'base-events and the p2a'base-events, we define AcceptorState with the SM2−class combinator of section 4. The transition functions for these two kinds of inputs, on_p1a and on_p2a, straightforwardly encode the updates to the state variables given in the pseudo-code in Figure 2 of [Ren11], and described in section 6.1.3.

We similarly factor the definition of Acceptor into classes AcceptorsP1a and AcceptorsP2a responding to the two kinds of inputs:

```
let Acceptor = AcceptorsP1a || AcceptorsP2a ;;
```

The definitions of these classes are similar, so consider the first:

```
class AcceptorsP1a =
  let f loc (ldr,_) bnum_acc = {p1b'send ldr (loc,bnum_acc)}
  in f o (p1a'base,AcceptorState) ;;
```

An AcceptorsP1a-event is a p1a'base-event. (It must be both an AcceptorState-event and a p1a'base-event, but every p1a'base-event is an AcceptorState-event.)

When the simple composition operator applies the local function f

- the loc parameter will match the location at which the input event occurs (the location of an instance of Acceptor)
- (ldr,_) will match the observation from p1a'base (the body of the incoming p1a message; so ldr is the location of the Leader who spawned the Scout that sent this message)
- bnum_acc will match the observation of AcceptorState (the state after the input event is processed)

Figure 20 Paxos Commander

```
class CommanderNotify bsp = Output(\ldr.p2a'broadcast accpts (ldr, bsp)) ;;

class CommanderState b =
  let tr loc (loc,b') waitfor =
    if b = b' then bag-remove (op =) waitfor loc else waitfor
  in
     SM1-class (\_.{accpts}) (tr, p2b'base) ;;

class CommanderOutput (b,(s,p)) =
  let f ldr (a,b') waitfor =
    if b = b'
    then if bag-size waitfor < threshold
         then decision'broadcast reps (s,p)
         else {} (* keep looking for majority *)
    else (* when b <> b', send preempted *)
         { preempted'send ldr b' }
  in
     Once(f o (p2b'base, CommanderState b)) ;;

class Commander bsp = CommanderNotify bsp || CommanderOutput bsp ;;
```

The output, {p1b'send ldr (loc,bnum_acc)}, directs to ldr a p1b message containing the location and state of the Acceptor that processes it.

Notice: AcceptorsP1a and AcceptorsP2a are not completely independent of one another. Each of them reacts to inputs by changing the state in a way that could affect the behavior of the other. That is why we use a single Moore machine AcceptorState, maintaining the whole state, to define the two Mealy machines responding to the two different kinds of inputs.

6.9 Commanders

A Leader spawns a Commander that tries to elect a particular proposal on a particular ballot. So we define a parameterized class Commander : PValue → Class(MSG). (Commander $\langle b, \langle s, p\rangle\rangle$) does two things:

```
class Commander bsp = CommanderNotify bsp || CommanderOutput bsp ;;
```

The CommanderNotify component sends a p2a message to all the Acceptors (the locations in accpts) and then terminates. The CommanderOutput component is a state machine whose input events are p2b messages (received in response to its initial p2a broadcast). It will send decision messages to all the Replicas or will send a preempted message to the Leader the spawned it; having done either, it terminates.

To define CommanderOutput we first define the "post" Moore machine CommanderState. One difference from the previous case is worth noting: Every AcceptorState-event is an Acceptor-event (i.e., results in a nonempty bag of directed messages); but there will be CommanderState-events that are not CommanderOutput-events.

For any b : Ballot_Num we define (CommanderState b) with SM1−class, since its input events are of just one kind, p2b messages. Its state is a bag of locations—the locations of all Acceptors from which it has not yet received a p2b message about ballot number b. Thus, it is initially accpts.

Consider the definition of (CommanderOutput $\langle b, \langle s, p\rangle\rangle$). The decision logic in figure 3(a) of [Ren11] is captured in the locally defined class f o (p2b'base,CommanderState b): It sends a preempted message if it receives an input with an acceptance for some ballot number other than b; broadcasts a decision message if it has received messages accepting ballot b from a majority of the acceptors; and otherwise makes no

Figure 21 Paxos Scout

```
class ScoutNotify b = Output(\ldr.p1a'broadcast accpts (ldr, b));;

class ScoutState b =
  let tr loc (loc,(b',r : PVlist)) (waitfor,pvalues) =
      if b = b'
      then let waitfor' = bag-remove (op =) waitfor loc in
           let pvalues' = append_news same_pvalue pvalues r in
             (waitfor',pvalues')
      else (waitfor,pvalues) in
   SM1-class (\_.{init_scout}) (tr, p1b'base) ;;

class ScoutOutput b =
  let f ldr (a,(b',r)) (waitfor,pvalues) =
    if b = b'
    then if bag-size waitfor < threshold
         then { adopted'send ldr (b,pvalues) }
         else {}
    else { preempted'send ldr b' }
  in
     Once(f o (p1b'base, ScoutState b));;

class Scout b = ScoutNotify b || ScoutOutput b ;;
```

output. Events falling through to the last case are CommanderState-events but not CommanderOutput-events. By applying the Once combinator to this class (which corresponds to the *exit* statements in the pseudo-code) we guarantee that there can be at most one CommanderOutput-event.

6.10 Scouts

A Leader spawns a Scout to get a particular ballot number adopted (if possible). So Scout : Ballot_Num → Class(MSG). Its definition is structured identically to that of Commander. It consists of a "notify" component that broadcasts p1a messages and an "output" component that tallies reponses to them. It will either send an adopted message to all Leaders or send a preempted message to the Leader that spawned it.

```
class Scout b = ScoutNotify b || ScoutOutput b ;;
```

Following the previous pattern we first define the "post" Moore machine ScoutState. The (ScoutState b)-events are p1b messages. The "relevant" messages, which cause state changes, are those that concern ballot b. The type of its state is List(Loc) × PVlist. The first component is the list of all Acceptors from whom it has not yet received a relevant message; the second is the list of all pvalues it has received in relevant messages.

In the definition of ScoutOutput from ScoutState the local function f captures the decision logic of Figure 3(b) of [Ren11], and also described in section 6.1.3.

6.11 Leaders

A Leader spawns a Scout and thereafter can spawn a Commander in response to a propose or adopted message, or it can spawn a Scout in response to a preempted message:

Figure 22 Paxos Leader

```
let on_propose loc ((s,p) : Proposal) (ballot_num, active, proposals) =
  let proposals' =
    if in_domain (op =) s proposals
    then proposals
    else add_if_new same_proposal (s,p) proposals
  in (ballot_num,active,proposals') ;;

let when_adopted loc (bnum,pvals) (ballot_num, active, proposals) =
  if bnum = ballot_num
  then let proposals' = update_proposals proposals (pmax (pvals : PVlist))
       in (ballot_num, true, proposals')
  else (ballot_num,active,proposals) ;;

let when_preempted ldr bnum (ballot_num, active, proposals) =
  if isl bnum & lt_bnum ballot_num bnum
  then let (r',loc') = outl bnum in (inl (r' + 1,ldr), false, proposals)
  else (ballot_num, active, proposals) ;;

class LeaderState =
  Memory3 (\l.{init_leader l})
          on_propose  propose'base
          when_adopted  adopted'base
          when_preempted  preempted'base;;

class LeaderPropose =
  let f loc (slt, p) (ballot_num, active, proposals) =
    if active & !(in_domain (op =) slt proposals)
    then {(ballot_num,(slt,p))}
    else {} in
    f o (propose'base,LeaderState);;

class LeaderAdopted =
  let f loc _ (bnum, _, props) = (map (\sp. (bnum,sp)) props)/~ in
    f o (adopted'base,LeaderState);;

class LeaderPreempted =
  let f ldr bnum (ballot_num, _, _) =
    if isl bnum & lt_bnum ballot_num bnum
    then {inl(fst(outl bnum) + 1,ldr)}
    else {} in
    f o (preempted'base,LeaderState);;

class SpawnFirstScout = OnLoc(\ldr.Scout(inl(0,ldr)));;

class Leader = SpawnFirstScout
           || ((LeaderPropose || LeaderAdopted) >>= Commander)
           || (LeaderPreempted >>= Scout) ;;
```

```
class Leader = SpawnFirstScout
             || ((LeaderPropose || LeaderAdopted) >>= Commander)
             || (LeaderPreempted >>= Scout) ;;
```

At any location *ldr*, SpawnFirstScout acts like the class (Scout `inl`$(\langle 0, ldr\rangle)$). (Recall that $\langle 0, ldr\rangle$ is the least ballot that the Leader at *ldr* owns.) To install an appropriate class at each location we use the primitive OnLoc introduced in section 6.5.6:

```
class SpawnFirstScout = OnLoc(\ldr.Scout(inl(0,ldr)));;
```

The state of a Leader, LeaderState, is a "pre" Moore machine defined with Memory3. The transition functions corresponding to its three kinds of input messages are on_propose, when_adopted, and when_preempted. The type of its state is Ballot_Num ∗ Bool ∗ (Proposal List). As noted in section 6.1.2, the Bool component of a Leader's state indicates whether the current Ballot_Num component is active or not; the (Proposal List) component is a consistent list of proposals, the proposals that the Leader is currently working on.

```
class LeaderState =
  Memory3 (\l.{init_leader l})
          on_propose   propose'base
          when_adopted   adopted'base
          when_preempted   preempted'base;;
```

We define the Mealy machines LeaderPropose, LeaderAdopted, and LeaderPreempted from LeaderState. Consider the last of these:

```
class LeaderPreempted =
  let f ldr bnum (ballot_num, _, _) =
    if isl bnum & lt_bnum ballot_num bnum
    then {inl(fst(outl bnum) + 1,ldr)}
    else {} in
    f o (preempted'base, LeaderState);;
```

By the definition of simple composition every LeaderPreempted-event must be both the arrival of a preempted message and a LeaderState-event (though the converse needn't be true, and in general won't be). But every event is a LeaderState-event.

When applying the local function f,

- ldr matches the location of the Leader at which the preempted event occurs;
- bnum matches the ballot number sent in the preempted message;
- (ballot_num, _, _) matches the state of the Leader when the message arrives; so, in particular, ballot_num matches the current ballot number.

Consider the condition in the conditional expression for f. The conjunct (isl bnum) is true when bnum is not dummy_ballot. In fact, an invariant of the protocol guarantees that this will always be true, but the test is included so that the declaration will typecheck statically, without knowing that invariant.[16] Thus, execution will take the then branch (and LeaderPreempted will spawn a Scout) iff bnum is greater than the ballot number in the Leader's state.

The only important fact about inl(fst(outl bnum) + 1,ldr), the ballot number passed to the spawned Scout, is that it belongs to the Leader at location ldr and is greater than bnum.

6.12 Replicas

In order to define a class that is a state machine, we have repeatedly used the strategy of defining a Mealy machine from a Moore machine. We do that again, but in a way that may seem backwards. ReplicaAux

[16] It's needed so that the subterm outl bnum will typecheck.

Figure 23 Paxos Replica

```
let out_tr tr loc x (s,_) = tr x s ;;

(* first_unoccupied ps = least positive integer that is *not* a member of ps *)
let first_unoccupied (ps : Slot_Num List) =
  list_accum (\a.\x. if x = a then a + 1 else a) 1 (quicksort-int ps)
;;

let propose p (rs, sn, prs, dcs) =
  if in_range (op =) p dcs
  then ((rs, sn, prs, dcs), {})
  else let s' = first_unoccupied (domain (prs ++ dcs)) in
       let prs' = add_if_new (op =) prs (s',p) in
       let msgs = propose'broadcast ldrs (s',p) in
         ((rs, sn, prs', dcs), msgs) ;;

let perform (cmd : Command) ((rstate, slot_num, proposals, decisions) : ReplicaState) =
  let (client,cid,ope) = cmd in
    if bl-exists decisions (\ (s,c) . s < slot_num & c = cmd)
    then ((rstate,slot_num + 1,proposals,decisions),{})
    else let (next, result) = apply_op ope rstate in
         let new_state = (next, slot_num + 1, proposals, decisions) in
           (new_state, {response'send client (cid,result)})
;;

let inner_tr p' state = (* applied when (n,p') \in decided *)
  let (rstate, slot_num, proposals, decisions) = state in
  let to_repropose = mapfilter snd (\ (m,p'') . m = slot_num & !(p'' = p')) proposals in
  let (new_state, proposes) = iterate_tr propose state to_repropose in
  let (new_state', responses) = perform p' new_state in
    (new_state', bag-append proposes responses) ;;

(* Each iteration of inner_tr performs one operation and also finds
 * all elements of proposals sharing the slot number of the operation
 * performed and reproposes them.  It does the reproposing first *)

let replica_decision v ((rstate, slot_num, proposals, decisions) : ReplicaState) =
  let decisions' = add_if_new (op =) decisions v in
  let ready = mapfilter snd (\ (s,_) . s = slot_num) decisions' in
    iterate_tr inner_tr (rstate, slot_num, proposals, decisions') ready ;;

class ReplicaAux =
  SM2-class (\_.{(init_replica,{})})
              (out_tr propose,          request'base)
              (out_tr replica_decision, decision'base) ;;

class Replica = (\_.snd) o ReplicaAux ;;
```

is a Moore machine whose internal state has type ReplicaState $*$ Message Bag. Replica is the projection of this onto the second component, returning only the messages:

```
class Replica = (\_.snd) o ReplicaAux ;;
```

To derive a Mealy machine from a Moore machine we must be able to compute the outputs from the input and the (resulting) state. Consider how a Replica acts. In response to an input it may take a sequence of steps, each of which changes the internal state of the Replica (by executing a command and by updating bookkeeping information) and sends a message. The pseudo-code of [Ren11] carries that out in a loop. A Replica event class must respond to an input by making a transition to the final state (at the end of the loop) and returning a bag that contains all the messages produced by that entire sequence of steps. We cannot compute that bag of messages just from the input value and the final state—at least, we cannot do so if the final state is simply a value of type ReplicaState. That is why the state type of ReplicaAux is ReplicaState $*$ Message Bag.

We use SM2−class to factor the definition of ReplicaAux in terms of its response to propose messages and to decision messages.

The auxiliary function out_tr lifts a transition function tr for a Mealy machine with inputs A, outputs Msgs, and internal state ReplicaState

$$tr : I \to \mathsf{ReplicaState} \to \mathsf{ReplicaState} \times \mathsf{Bag}(\mathsf{Msgs})$$

to an equivalent transition function out_tr tr for a Moore machine with inputs I and internal state ReplicaState $\times$ Bag(Msgs):

$$\mathsf{out\_tr}\ tr : \mathsf{Loc} \to I \to \mathsf{ReplicaState} \times \mathsf{Bag}(\mathsf{Msgs}) \to \mathsf{ReplicaState} \times \mathsf{Bag}(\mathsf{Msgs})$$

The initial argument of type Loc is ignored. It's included so that out_tr tr will have the type required of an argument to SM2−class.

The operation

$$\mathsf{propose} : \mathsf{Proposal} \to \mathsf{ReplicaState} \to \mathsf{ReplicaState} \times \mathsf{Bag}(\mathsf{Msgs})$$

encodes the *propose* operation of [Ren11], which a Replica performs in response to a propose message. Thus (out_tr propose) lifts this to a transition function for ReplicaAux. (Note: propose invokes first_unoccupied , which finds the first empty slot in a list of proposals by a logically correct, but woefully inefficient procedure; it begins by sorting its input. There's no point in defining a more efficient one, because a real implementation would implement propose differently, to avoid accumulating an unbounded list of proposals in the state of the Replica.)

The operation

$$\mathsf{perform} : \mathsf{Command} \to \mathsf{ReplicaState} \to \mathsf{ReplicaState} \times \mathsf{Bag}(\mathsf{Msgs})$$

encodes the *perform* operation of [Ren11]. In response to a decision message a Replica may invoke this repeatedly in a loop: inner_tr does one iteration of the loop; (iterate_tr inner_tr) executes the loop, accumulating all the messages to be sent; and replica_decision encodes the entire response. Thus (out_tr replica_decision) lifts this to the other transition function for ReplicaAux.

```
class ReplicaAux =
  SM2−class (\_.{(init_replica,{})})
              (out_tr propose,          request'base)
              (out_tr replica_decision, decision'base) ;;
```

Figure 24 Paxos main program

```
main Leader @ ldrs || Acceptor @ accpts
```

7 Definitions of combinators

General simple composition

Section 3.1 introduces the simple composition combinator. Given n classes X_1, ..., X_n, of types T_1, ..., T_n respectively, and given a function F of type $\texttt{Loc} \to T_1 \to \cdots \to T_n \to \texttt{Bag}(T)$, one can define the class F `o` $(X_1, \cdots, X_n)$. This combinator is defined in terms of one of the Logic of Events' primitive combinators. Given n classes X_1, ..., X_n, of types T_1, ..., T_n respectively, and given a function F of type $\texttt{Loc} \to \texttt{Bag}(T_1) \to \cdots \to \texttt{Bag}(T_n) \to \texttt{Bag}(T)$, the class F `O` $(X_1; \cdots; X_n)$ is one of the Logic of Events' primitive combinator. The class F `o` $(X_1, \cdots, X_n)$ is defined as:

$$(\lambda loc.\ \lambda b_1 \ldots . \lambda b_n.\ \bigcup_{x_1 \in b_1} \cdots \bigcup_{x_n \in b_n} F\ loc\ x_1\ \cdots\ x_n)\ \texttt{O}\ (X_1; \cdots; X_n)$$

Until

The binary infix operator until can then be defined in terms of this more general simple composition combinator as follows:

```
import bag-null;;

class until X Y =
  let F loc b1 b2 = if bag-null b2 then b1 else {}
  in F o (X, Prior(Y)) ;;

infix until;;
```

The bag-null function is a function that returns true iff its argument is the empty bag. Note that using an infix declaration, one can declare infix operators in EventML.

Once

The Once operator can be defined in terms of the until operator as follows:

```
class Once X = (X until X) ;;
```

Output

The Output operator can be defined in terms of the Once operator as follows:

```
class Output b = Once(b o ());;
```

The "at" combinator

The binary infix operator @ can be defined in terms of the simple combinator as follows:

```
import bag-deq-member ;;
class @ X locs =
  let F loc x = if bag-deq-member (op =) loc locs then {x} else {}
  in F o X ;;
infix @ ;;
```

(Note that this code is not valid EventML code because @ is not a valid identifier.)

Parallel combination

The parallel combinator can be defined in terms of the more general simple combinator as follows:

```
class || X Y = (\loc.\b1.\b2.b1++b2) o (X,Y) ;;
infix || ;;
```

(Note that this code is not valid EventML code because || is not a valid identifier.)

Disjoint union

The *disjoint union* class combinator. X (+) Y is a class of type A + B that recognizes both X-events and Y-events. The observations made by X are tagged with inl and the observations made by Y are tagged with inr:

```
class X (+) Y = ((\_.\x.{inl(x)}) o X) || ((\_.\x.{inr(x)}) o Y) ;;
```

If (and only if) e is both an X-event and a Y-event, X (+) Y(e) is a bag with two elements.

SM1-class, . . . , Memory1, . . .

For any n, the combinators SMn class and Memory$n$ are defined in section 4.

8 Configuration files

Parameters to our specifications are of two kinds. Some are "abstract"—e.g., the integer parameters threshold (see section 3.2) and flrs (see section 5). We can instantiate these by providing a Nuprl term of type integer. Others are "real world"—e.g., the parameter client of type location. Their meanings are specific to a particular installation of EventML: the messaging system determines what must be supplied to instantiate a location parameter. Our prototype assumes that messaging is by TCP/IP, and a location is a pair consisting of an IP address and a port.[17]

The parameter declarations

```
parameter nodes  : Loc Bag ;;
parameter client : Loc ;;
parameter uid    : Loc → Int ;;
```

illustrate the open-ended nature of real world parameters.

Suppose that we supply an (IP address, port) pair for client and a list of such pairs for nodes.[18] How do we instantiate uid? Knowing the locations, we could simply define a function that assigns integers to them. If we wanted a more flexible implementation, we might want to base uid on the MAC address of a node's network card; in that case the configuration file would provide some reference to a piece of code that does the computation. For now, the only primitive real-world type that we allow is Loc. All other parameter types must be intepretable from Loc and abstract types.

Here is what a configuration file looks like:

[17] Note that TCP/IP provides stronger guarantees—namely, FIFO delivery—than our examples have assumed.
[18] Computationally, a bag is just a list in which we ignore the order.

```
%locations
 n1: 192.168.0.12 19777
 n2: 192.168.0.13 19778
 n3: 192.168.0.14 19779

%parameters
 nodes:  {LOC(n1);LOC(n2);LOC(n3)}
 client: LOC(client)
 uid:    \l.if l = LOC(n1) then 1 else if l = LOC(n2) then 2 else 3

%messages
 n1: (``config``, Int * Loc, (1, LOC(n2)))
 n2: (``config``, Int * Loc, (1, LOC(n3)))
 n3: (``config``, Int * Loc, (1, LOC(n1)))
 n2: (``choose``, Int, 1)
```

This is an example of a configuration file for the leader election in a ring protocol presented in section 3.3. A configuration file is divided into three parts: the locations part declares the machines on which one wishes to install the specified protocol (n1 is a location name which is specified by the IP address 192.168.0.12 and the port number 1977); the parameters part instantiates the parameters declared in the given specification (the leader election in a ring specification presented in section 3.3 declares three parameters: nodes, client, and uid); the messages part declares a bag of messages initially in transit. One has to declare at least one message in transit because EventML allows on to define reactive agents that can only react on receipt of messages. Therefore nothing happens as long as no message is received.

9 EventML's syntax

Identifiers

An identifier can either be *alphanumeric* or *symbolic*. An alphanumeric identifier is a sequence of letters, digits, primes (quotes), dashes and underscores starting with a letter. For example, bag_map, bag−map, bag−map', and bag−map1 are identifiers, but 1bag−map is not. A symbolic identifier is a sequence of the following symbols: !, %, &, #, /, <, =, ?, \, ~, ^, |, >, −, :, +, @, ∗. Some alphanumerics as well as some symbolic identifiers are disallowed because they are reserved keywords. They are described in Figures 25 and 26 bellow.

Let Vid be the set of identifiers and let *vid* range over identifiers.

Type variables

A type variable is an alphanumeric identifier preceded by primes (quotes). For example, 'a and ''a are type variables

Let TyVar be the set of type variable and let a range over type variables.

Character sequences

Let CharSeq be the set of sequences of characters other than backquotes (`) and let *cseq* range over CharSeq.

Other syntactic forms

Figures 25 and 26 defines EventML's syntax. In this figure we write $\ulcorner x \urcorner$ to indicate that x is optional. These brackets are not part of EventML's syntax. For example, a program *prog* can either be a declaration followed by two semicolons, or a declaration followed by two semicolons followed by another program (it allows us to define recursive production rules).

We also impose the following syntactic restrictions:

- In a program, a declaration of the form `specification` *vid* has to be the first declaration and there can only be one.

Figure 25 EventML syntax – expression

n	$\in$	Nat		(natural numbers)
ptc	$\in$	PostTyC	::=	`Int` | `List` | `Bool` | `Unit` | `Bag` | `Class` | `Msg` | `Loc` | `Token`
itc	$\in$	InfTyC	::=	$*$ | $\rightarrow$ | $+$
b	$\in$	Bool	=	`true` | `false`
op	$\in$	Op	::=	`+` | `-` | `*` | `/` | `=` | `.` | `++` | `<` | `>` | `or` | `&` | `>>=` | `||` | `@`
$atexp$	$\in$	AtExp	::=	vid | n | b | $\sim atexp$ | `inl`(exp) | `inr`(exp)
			|	$(exp_1, \ldots, exp_n)$
			|	$\{exp_1; \ldots; exp_n\}$
			|	`Prior`(exp)
			|	`Once`(exp)
			|	`Output`(exp)
			|	`OnLoc`(exp)
			|	`(`exp`)`
exp	$\in$	Exp	::=	$atexp$
			|	$atexp$`/`$\sim$
			|	exp `o` $(exp_1, \cdots, exp_n \ulcorner$, `Prior(self)?`$exp' \urcorner)$
			|	exp_1 op exp_2
			|	exp`:`ty
			|	exp $atexp$
			|	$\backslash pat$ `.` exp
			|	`if` exp_1 `then` exp_2 `else` exp_3
			|	`let` $bind$ `in` exp
			|	`letrec` $bind$ `in` exp
			|	`class` $bind$ `in` exp
			|	$atexp$ `where` $bind$
pat	$\in$	Pat	::=	vid
			|	`_`
			|	$(pat_1, \ldots, pat_n)$
			|	pat`:`ty
$tyseq$	$\in$	TySeq	::=	ϵ
			|	ty
			|	$(ty_0, \ldots, ty_n)$
ty	$\in$	Ty	::=	a
			|	$tyseq$ ptc
			|	ty_1 itc ty_2
			|	(ty)

- In a program, a declaration of the form `main` *exp* has to be the last declaration and there can only be one.
- In an expression of the form `letrec` *bind* `in` *exp′*, where *bind* is of the form *vid* $atpat_1 \cdots atpat_n$ `=` *exp*, either $n >= 1$ or *exp* is a lambda expression the form $\backslash pat$ `.` exp'' (i.e., a recursive declaration can only bind a function).
- In a declaration of the form `letrec` *bind*`;;`, where *bind* is of the form *vid* $atpat_1 \cdots atpat_n$ `=` *exp*, either $n >= 1$ or *exp* is a lambda expression (i.e., of the form $\backslash pat$ `.` exp').

References

[ABC+06] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. Innovations in computational type theory using nuprl. *J. Applied Logic*, 4(4):428–469, 2006.

Figure 26 EventML syntax – declarations

tok	∈	Token	::=	`` `cseq` ``
toks	∈	Tokens	::=	$[tok_0; \ldots; tok_n]$
status	∈	Status	::=	`internal`
			|	`input`
			|	`output`
hdropt	∈	HdrOpt	::=	`base` *vid*
			|	`send` *vid*
			|	`broadcast` *vid*
hdropts	∈	HdrOpts	::=	$hdropt\ulcorner, hdropts\urcorner$
header	∈	Header	::=	$status\ (toks : ty\ulcorner, hdropts\urcorner)$
bind	∈	Bind	::=	$vid\ atpat_1 \cdots atpat_n$ `=` *exp*
dec	∈	Dec	::=	`let` *bind* `;;`
			|	`letrec` *bind* `;;`
			|	`class` *bind* `;;`
			|	`parameter` *vid* `:` *ty*
			|	`import` $vid_0 \cdots vid_n$
			|	`MSGS` $header_0 \cdots header_n$
			|	`main` *exp*
			|	`specification` *vid*
prog	∈	Prog	::=	$dec\ \ulcorner prog\urcorner$

[BC08] Mark Bickford and Robert L. Constable. Formal foundations of computer security. In *NATO Science for Peace and Security Series, D: Information and Communication Security*, volume 14, pages 29–52. 2008.

[BCG10] Mark Bickford, Robert Constable, and David Guaspari. Generating event logics with higher-order processes as realizers. Technical report, Cornell University, 2010.

[BCH+00] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *In DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–160. IEEE Computer Society Press, 2000.

[Bic09] Mark Bickford. Component specification using event classes. In *Component-Based Software Engineering, 12th Int'l Symp.*, volume 5582 of *LNCS*, pages 140–155. Springer, 2009.

[BKR01] Mark Bickford, Christoph Kreitz, and Robbert Van Renesse. Formally verifying hybrid protocols with the nuprl logical programming environment. Technical report, Cornell University, 2001.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[CHP84] Guy Cousineau, Gérard Huet, and Larry Paulson. *The ML handbook*, 1984.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Hay98] Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, Department of Computer Science, 1998. Technical Report TR98-1662.

[KHH98] Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communication systems. In *Automated Deduction - CADE-15, 15th Int'l Conf. on Automated Deduction*, volume 1421 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 1998.

[KR11] Christoph Kreitz and Vincent Rahli. *Introduction to Classic ML*, 2011.

[Kre02] Christoph Kreitz. *The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 2002. `http://www.nuprl.org/html/02cucs-NuprlManual.pdf`.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Ler00] Xavier Leroy. *The Objective Caml system release 3.00*. Institut National de Recherche en Informatique et en Automatique, 2000.

[LKvR+99] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth P. Birman, and Robert L. Constable. Building reliable, high-performance communication systems from components. In *SOSP*, pages 80–92, 1999.

[Ren11] Robbert Van Renesse. Paxos made moderately complex. `www.cs.cornell.edu/courses/CS7412/2011sp/paxos.pdf`, 2011.

[Win88] Glynn Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, volume 354 of *LNCS*, pages 364–397. Springer, 1988.

Index